A SHORT COURSE IN
RESEARCH DATA MANAGEMENT
USING / SAS /

PREPARED FOR THE

INSTITUTO INTERAMERICANO DE CIENCIAS AGRICOLAS - OEA

BY

RONALD W. HELMS, PH.D.

DEPARTMENT OF BIOSTATISTIC

UNIVERSITY OF NORTH CAROLINA

CHAPEL HILL NC 27514, U.S.A.

JANUARY - FEBRUARY, 1980

PREFACE: BACKGROUND AND OBJECTIVES

Summary:

There exists a body of knowledge - techniques and strategies for managing data from scientific research - called Research Data Management (RDM). The primary objective of the course is to present an introduction to the main techniques and strategies of RDM.

RDM is a very applied art; the methods make sense only if applied in practice. To apply the techniques one must use appropriate computer software. At present, the best available software for this purpose is SAS. Although the audience for this course may have some familiarity with SAS we assume very little background in the use of SAS for modern RDM. Therefore, the first part of the course is an introduction of SAS programming, with special emphasis on SAS facilities which are particularly useful in RDM.

In summary, then, the objectives are: (1) to present SAS programming techniques useful for managing research data and (2) to present the main techniques and strategies for managing research data.

The advent of the electronic computer led very quickly to the creation ofnew disciplines, including computer science, systems analysis, and various combinations and subsets of these disciplines concerned specifically with the efficient and secure management and processing of data. Both the scientific and commercial communities were quick to take advantage of new computer technology in the 1950s. Commercial users, recognizing that data processing, per se, accounted for a great part of the cost of computer operations, supported the study of efficient techniques and systems for processing commercial data. Producers of highly efficient software for processing commercial data were quickly rewarded. For an example one need only look at the success of companies producing efficient sorting packages.

The scientific community generally took a different direction. In the 1950s and 1960s the most important scientific computing problems were a lack of computational software. Great resources were applied to develop general scientific software including, in the statistical area, the large packages such as BMD, SPSS, BMDP and SAS72. In the early to mid 1970s practitioners in the field of 'statistical computation' began to realize that the ability of scientists to collect data far outstripped the ability of computing personnel to effectively process and manage the data. Once data were processed into a suitable format the statistical packages could easily produce most of the required statistics.

From that situation a new sub-discipline, Research Data Management, was born. Research Data Management (quickly abbreviated RDM) consists of the study of the problems peculiar to managing and processing data from scientific research, together with the strategies and methods developed for solving those problems.

Some of the techniques of RDM are straighforward adaptations of commercial data management and systems analysis methods. Sorting scientific data is much the same as sorting commercial data, for example, and the same software is

used for both. However, there are important differences between research data and commercial data which require different data management strategies.

One of the important differences is the effect of data errors. For example, compare a banking system for processing checking account transactions versus a 'system' for processing data from a large social science survey. A random error rate of 1% in the checking account system would be intolerable; the same error rate in the social science survey might be literally impossible to detect. Thus, quality control strategies are different for commercial and research data management.

A second important difference involves the sizes and 'shapes' of datasets. Commercial datasets (as the checking account case, for example) often have relatively few 'variables' or 'fields' (typically 6 or 10 fields for checking data) but a very large volume of records to be processed. In contrast, scientific data frequently have dozens, or hundreds of variables and fewer records ("cases") than a "comparable" commercial dataset.

Research Data Management saw several important advances in the 1970s. There were important software developments, notably the data management capabilities of SAS72, SAS76 and SAS79. There were also researchers and teachers who began to develop new techniques (as, for example, table driven systems, inventory subsystems and 'statusbytes', all the Lipid Research Clinics 'Visit 2 Data Management System' (Helms, 1972) and organize and consolidate knowledge in the area. Examples of the latter include Bob Teitel at the Urban Institute, who presented a series of papers at the Interface Symposia and the present author and Wendell Smith at the University of North Carolina, who developed a graduate subcurriculum in Research Data Management.

As a result of the efforts of these and other researchers, in 1980 RDM is beginning to be recognized as a bona fide discipline, generally within the area

of statistical computation. There exists a body of knowledge, a collection of techniques and strategies, for solving the important problems facing the manager of research data.

The primary objective of this short course is to present an introduction to the most important RDM techniques and strategies. RDM is inherently a very practical and applied science/art, however, and effective presentation requires that the student practice the techniques and strategies with real software on realistic data. At this point the best generally available software for the management of small or moderate data files is found in SAS79.

Therefore, we have chosen to present, as a part of the text, a brief short course in SAS programming with emphasis on commands and procedures important to RDM.

I. INTRODUCTION

1. An Overview of Research Data Management:

What is it?

Research Data Management (RDM) is a collection of methods and strategies for managing data from scientific research. This definition is straightforward, but to understand what RDM really is, one must know something about managing data with computers and some of the factors which distinguish research data from, say, commercial data. These topics constitute some of the substance of the latter part of this text.

The activities of RDM can be classed into five general categories:

1. Planning
2. Design and Implementation of RDM Systems
3. "Production" Data Processing
4. Analysis File Data Processing
5. Statistical Computation

These categories are approximately in the sequence in which activities are performed for a particular project. Planning is obviously a management function and includes estimation of needed resources (funds, personnel, calendar time), acquisition of resources, scheduling activities, etc. (Helms, 1978). Design and implementation involves RDM System Analysis for the design phase and the efforts of RDM trained programmers for implementation (programming, debugging, testing, documenting).

Once an RDM system is implemented it is typically used, or operated, by data processors, persons with the personality and skill to handle data carefully, to input the data to the computerized RDM system and use the system to detect and correct errors and produce well-organized, 'clean' master

files. This is 'production' data processing. Production data processing work tends to be routine, tedious work.

The master files resulting from production data processing are usually ill-suited for statistical analysis, in part because an RDM system is usually designed to optimize the production data processing phase. In addition, master files may have a structure which is too complicated for direct use by statistical analysis programs. Before statistical computation can be performed (phase 5), one must usually perform several tasks to produce analysis files from master files: this is the "Analysis File data processing" phase.

Finally, one has files which can be input directly to a statistical package such as SAS, to perform the desired statistical computations, the last phase of the RDM process.

This discussion is, of course, very general. In a particular project, one or more of these phases may be so 'easy' that it is essentially incorporated into other phases. In other, larger, more complex projects, every phase may have a significant cost and require a significant interval of calendar time.

Topics to be Discussed:

There is, clearly, far more material in the topic of RDM than can be covered in a text of this magnitude. The topics to be covered here are:

1. SAS Programming Techniques
2. Components of a General RDM System
3. Designing an RDM System
4. Planning for RDM

RDM is such an applied art/science that to understand the techniques the student must necessarily practice them. This practice requires the use of appropriate computer software and at this point SAS provides the best generally available RDM software for general projects of small-to-moderate size. We do not assume the student has a background in SAS, particularly those parts of SAS which are particularly useful for RDM. Thus we begin with a short course in SAS-RDM techniques. This 'sub-shortcourse' actually provides a fairly comprehensive review of SAS programming facilities and relies heavily upon readings in SAS79. (SAS79 is our abbreviation of SAS User's Guide, 1979 Edition).

In the second part of the course a general RDM system flowchart is presented and each of the component subsystems is briefly discussed.

The third major topic, RDM System Design, has been omitted from this short text. In part, this is due to the shortness of the course and, in part, to the fact that RDM system design is still a fairly primitive art.

Planning for RDM is discussed briefly in Appendix I. RDM planning is very similar to planning for other types of software and the material in this section relies heavily on the work of other software managers(Brooks (1975), Metzger (1973).

II. SAS PROGRAMMING FOR RDM

1. Overview of SAS Processing:

In spite of the name, 'Statistical Analysis System', SAS is really a data processing language with extensive statistical analysis facilities. SAS is different from most other programming languages such as Fortran and PL/I. One difference is that there are basically two types of SAS 'programs'. One type is named a "DATA Step" because it begins with a SAS DATA statement and is specifically oriented to creating a new "SAS dataset" (a dataset in a special format). The other type is called a "PROC STEP" because it begins with a SAS PROC (procedure) statement. Most PROC steps involve SAS subprograms to perform some type of statistical computations. Since this is a text about data management, the principal interest will be in DATA steps.

A DATA Step contains a program for inputting one or more datasets and outputting (usually) a SAS dataset. The program statements superficially appear much like PL/I statements. The same symbols are used, for the most part, for numerical operators such as multiplication and division; and each statement end with a semicolon (";"). But, the SAS DATA step program is structurally different.* The basic mode of operation is for SAS to input a data record (an observation from a SAS dataset or one or more data cards from an external dataset), execute the program statements once, then automatically write the SAS "observation" to the output SAS dataset. The SAS programmer is not required to write statements to read and

---

* It is an interesting comment on "modern" life that, as this is being written the author is an airplane bound from Costa Rica to Guatemala, he is thrilled by the fact that for the first time he can simultaneously see both the Atlantic and Pacific Oceans, from the right and left sides of the plane, respectively.

write data records -- all this is performed semi-automatically by SAS. The programmer can of course, take control of these functions to almost any extent desired.

Since (input/output) is typically one of the most bothersome aspects of programming and since SAS takes care of all I10 from and to SAS datasets, SAS can be an extremely easy and efficient language for processing data once those data are initially stored in SAS datasets. Moreover, SAS provides substantial facilities to incorporate data documentation directly in SAS datasets. These and other features make SAS approximately an order of magnitude better than traditional programming languages for managing research (and other) data.

1.1 OS Datasets, SAS Databases, and SAS Datasets

The basic purpose of this section is to eliminate, as quickly as possible, confusion which arises from multiple uses of the work dataset. In general terms, a dataset is any set of data. However, we will be concerned with particular types of datasets.

An OS dataset is a dataset which is accessed via the Operating System:

Examples include:

1. a deck of cards in a "SYSIN" dataset
   //SYSIN DD *
   deck of cards = OS dataset
2. A collection of records on a magnetic tape.
   (//tapeset DD Disp = OLD, UNIT = TAPE, DSN = ABC.DEF)
3. A collection of records on a magnetic disk
   (//DISKEX DD DISP = OLD, UNIT = DISK, DSN = QRS,TUV)
4. A "SYSOUT" dataset is really a disk dataset. After the job has completed execution the operating system copies each SYSOUT dataset to the printer and then deletes the dataset from the disk, (Example: //SYSPRINT DD SYSOUT = A,DCB=(RECFM=VBA, BLKSIZE = 200)

A dataset may contain data in any of several formats; we are primarily

interested in two formats:

"EBCDIC" ("Extended Binary Coded Decimal Interchange Code)
Cards are "punched", basically, in this format,
SYSOUT datasets are in EBCDIC
"Internal" or "hexadecimal floating point" format,
used by SAS for its datasets.

The computer does not perform arithmetic on numbers stored in EBCDIC code. In SAS, numbers must be converted from EBCDIC (or other code) to "internal hexadecimal floating point code" or "internal format" before being processed as numbers. The conversion process is often relatively expensive.

SAS converts data from EBCDIC to an internal format and then writes the data onto disk (or tape) in a very special format.

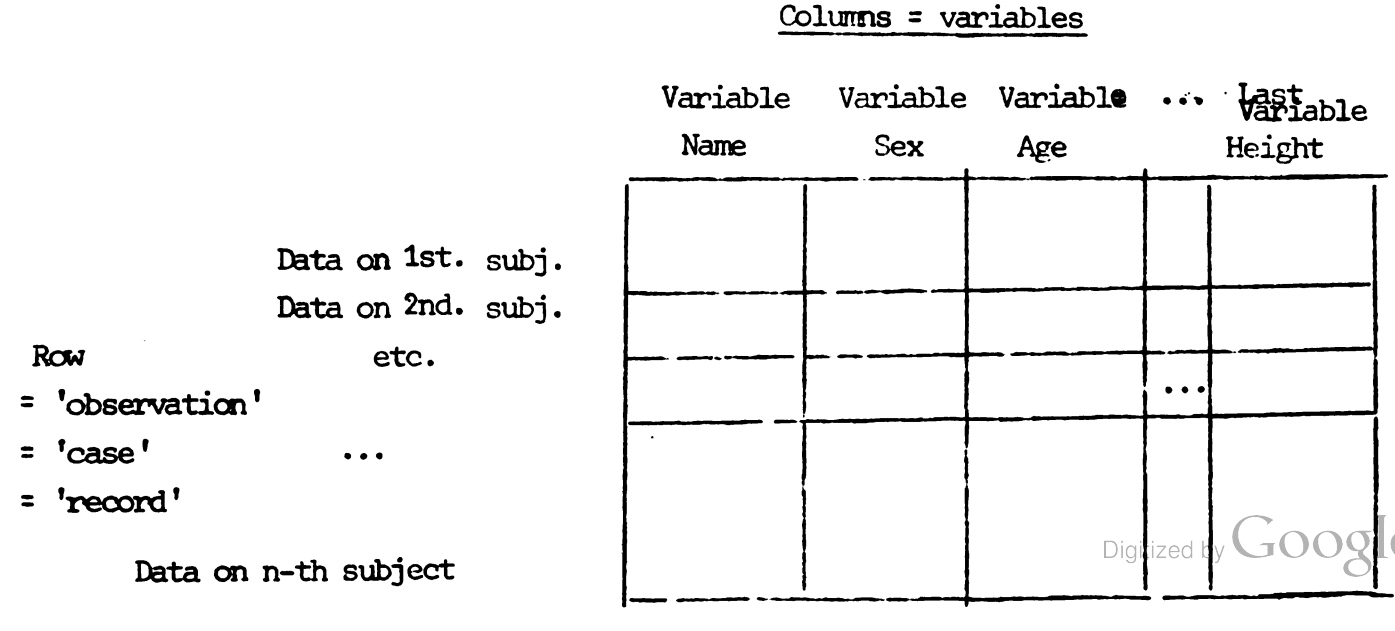
A SAS Database is an OS dataset (on disk, sometimes on tape) written by SAS in a special format. A SAS Database contains:

*One or more SAS Datasets (discussed below)

*Information about each SAS Dataset in the database. (We'll discuss this later)

A SAS dataset consists of a data matrix plus special information on each variable.

A standard data matrix (also called a "flat file") is an array in the following format:



The SAS dataset contains one data matrix (plus info such as names of variables, formats, etc.)

SAS creates (writes) a SAS dataset one entire observation at a time (one row at a time). SAS reads a SAS dataset one observation at a time (all variables at once).

Within a SAS dataset every observation has the same structure. The variables are in the same order. A variable has the same length (number of bytes) used for storage throughout all observations. Only data values differ from one observation to the next.

SAS can put several SAS datasets in one SAS database. The datasets may be completely different, i.e., they may have no particular relationship to one another. Of course, they may also be closely related.

To refer to a SAS dataset within a SAS database one must tell SAS two things:

*The information to locate the database (i.e., the OS dataset containing the database). This is a ddname on a DD statement. For example, in the following OS JCL statement,
//PERM DD DISP = OLD, UNIT = DISK,/DSN = ABC. DEF
"PERM" is the SAS Database name = ddname and
"ABC.DEF" is the OS DSN (tells OS where to find the dataset).
*The name of the SAS dataset in the database: e.g., in the SAS Statement
DATA PERM.MYDATA;
"PERM" is the database name (=ddname)
"MYDATA is the name of the SAS dataset within the database PERM.

(We will ignore the use of dataset specifications of the form DATA MYDATA, i.e. those without a database name, because their use is poor programming practice).

There is a special, temporary SAS database called WORK. WORK is deleted, along with any SAS datasets stored in it, at the end of the job. WORK is useful for storing temporary results. The name of a dataset in WORK is like names of datasets in other databases, e.g.: WORK. MYDATA.

1.2 Overview of Processing.

SAS is a collection of a large number of computer programs and a 'supervisor' program. The supervisor calls the various other programs as they are needed. A typical SAS job contains several "SAS steps". A DATA statement or a PROC statement is always the first statement of a SAS Step. A SAS "Step":

Begins with a PROC or DATA statement and ends at the last statement, or the last statement before the next PROC or DATA statement. This structure is illustrated by the following:

```
                                //SYSIN  DD*
----------------------------------------------------------
DATA           }                          First step
     Statements}
DATA           }                          Second step
     Statements}
PROC     }                                Third step
     BY  }
DATA  }                                   Fourth step
----------------------------------------------------------
```

In processing a SAS job, the supervisor:

1. Identifies the first step;
2. Calls the compiler to translate SAS statements into computer-executable statements (mostly subroutine calls);
3. Transfers control to the compiled program
4. If an error arises in the execution of the compiled program, the supervisor takes control and decides whether to: (1) terminate processing, or (2) continue processing with OBS = o (to be explained), or (3) continue normal processing
5. (Normal processing): Print "NOTES" about the step execution
6. (Normal processing): Identify the next step and go to 2 above.

A 'DATA step' (step beginning with a DATA statement) typically involves the creation of a new SAS dataset (within a specified SAS database). The input data may come from an external (EBCDIC) OS dataset or from one or more SAS datasets in specified SAS databases.

A "PROC step" (step beginning with a PROC statement) usually involves analysis of a SAS dataset (which was created earlier). The result is usually printed output rather that new SAS datasets.

We will have many exceptions: some DATA steps are "report generating programs' which produce a printout as the main output. Some PROCs, such as PROC SORT, principally process data.

1.3 SAS Execution of Data Steps

After compiling a program from a DATA step, the SAS supervisor executes the program once for each observation in the data. After executing all the statements in the program, SAS writes the observation, in 'internal format', to the output SAS dataset. A "typical" data input phase program may be diagrammed as follows:

```
DATA outputdsn;
          Declaration, such as LABEL, FORMAT
INFILE
INPUT

          computation and logical statements
```

Typically, SAS would repeat the following sequence for each input observation:

a. Execute the INPUT statement (input the variables listed in the statement, according to the specifications contained in the statement.)
b. Execute the computation and logical statements.
c. At the 'bottom' or 'end' of the program SAS would write out the observation (values of all the variables for this observation) into the SAS dataset named in the DATA statement.

The whole sequence is repeated for each observation. As we will see later, the programmer can change this sequence in many ways. However, unless changed by the programmer, SAS uses the sequence indicated above.

2. The Data Input Phase (DIP)

The first computerized phase of data processing is the Data Input Phase, which has the following objectives:

1. To input data from an "external" (EBCDIC, card, or tape) OS dataset and create a SAS dataset within a specified SAS database.
2. To perform preliminary checks for gross errors such as missing cards, missing values, etc.
3. To create internal documentation of the dataset such as names of variables, extended variable labels, formats of variables, etc.

The basic flowchart of a data input phase program, shown in Figure 2.1, is very simple. Any complicated data manipulations are postponed until after the data are stored in a SAS dataset. Because of the substantial amount of documentation, SAS programs for the data input phase are often long, but straightforward.

2.1 A SAS Data Input Phase Example

An example of a SAS Data Input Phase program is shown in Figure 2.2. The format of the input data, and a listing of a few cards are shown in Figure 2.3. In this example the data format is very simple: there is one card per observation and the card format is fixed.

The example is fairly typical of most DIP SAS programs except for the fact that most programs have more complicated data input. The following paragraphs discuss Data Input Phase programming, and related SAS statements, in terms of the example. Extension of the ideas to the more general case is straightforward.

Job statement. The details of the JCL JOB statement have been omitted. The JOB statement varies markedly from installation to installation. The reader should obtain local JCL details from computer installation officials.

Figure 2.1

The Simple Flowchart of a Data Input Phase

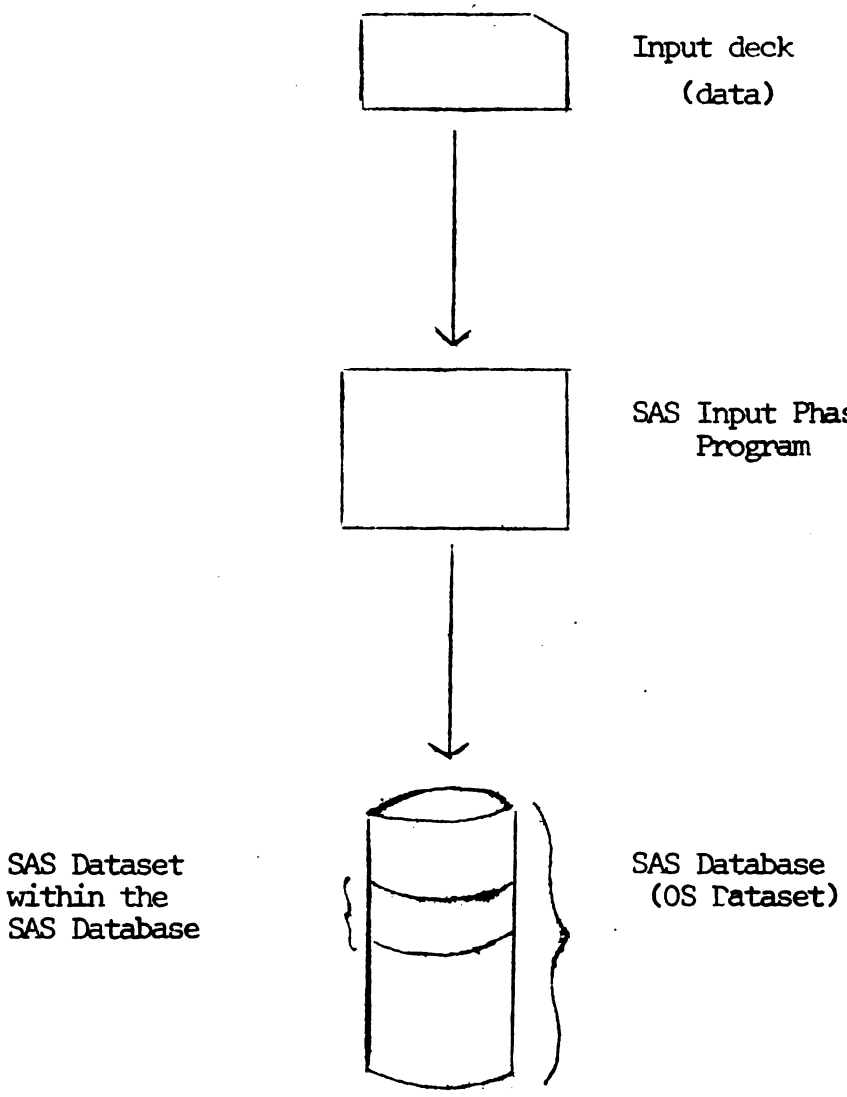


Figure 2.2 An example of a SAS DATA Input Phase Program
(The data format is defined in Figure 2.3)

```
//                      JOB
//                    EXEC SAS
// PERM DD  DISP=OLD, UNIT=DISK, DSN=ABC.DEF.GHI
//*PERM WAS CREATED IN AN EARLIER JOB.
// TARJETAS  DD*
                              data cards
// SYSIN     DD*
TITLE EXAMPLE OF SAS DATA INPUT PHASE;
TITLE2 STORE PHYSICAL FITNESS DATA IN PERM.PFD79;
DATA PERM.PFD79  (LABEL=1979 PHYSICAL FITNESS DATA);
     INFILE TARJETAS;

     LENGTH              NAME                $ 20
                         SEX                 $  1
                         HEIGHT
                         WEIGHT                 4;
     LABEL               HEIGHT=EIGHT OF SUBJECT IN CM
                         WEIGHT=SUBJECT WEIGHT IN KG
                         AGE=SUBJECT AGE IN YEARS
                         ;
     FORMAT              DATEBORN
                         SURVDATE    YYMMDD8;
     INPUT               CARDTYPE            $ 1 - 5
                         IDNUM                 6 - 9
                         NAME                $11 - 30
                         SEX                 $32
                         HEIGHT              $34 - 38
                         WEIGHT               40 - 44
          46             DATEBORN    YYMMDD6.
          53             SURVDATE    YYMMDD6.;
       AGE =             (SURVDATE-DATEBORN)/365.25;
     PROC PRINT
                  TITLE3  LISTING OF INPUT DATA;
     PROC SORT; BY NAME;
     PROC PRINT; ID NAME;
                  TITLE3  LISTING OF INPUT DATA SORTED BY NAME;
```

Figure 2.3 PHYSICAL FITNESS DATA FORMAT (PFD79) and PARTIAL DATA LISTING

Format

Cols	Description	
1 - 3	'PFD'	Card type descriptor (memonic)
4 - 5	'79'	Year the data were collected
6 - 8	OBS.	Get data from staff, but make up names
6 - 9	IDNUM	Identification number of the subject
11 - 30	NAME	Subject's name (Last, first, initial)
32	SEX	Coded as 'M' or 'F';
34 - 38	HEIGHT	Subject's height, in cm
40 - 44	WEIGHT	Subject's weight in Kg (xxx.x)
46 - 51	DATEBORN	Subject's date of birth, (yymmdd)
53 - 58	SURVDATE	Date on which survey was taken (yymmdd)

PERM DD Statement. This statement identifies the SAS database which will contain the SAS dataset created by the program. As with the JOB statement, DD statement details vary greatly from installation and the reader is again directed to local computer officials to determine the mechanism for creating permanent disk datasets and local JCL for using them.

TARJETAS DD Statement. The SYSIN dataset with ddname TARJETAS contains the input data for the program. Data may also be placed inside the SAS program, preceeded by a "CARDS"; SAS statement. We feel it is good programming practice to separate the data from the program, as has been done here. It is often the case that one tests a program with a small deck of data cards and, once testing is completed, a large card-image dataset on tape or disk is input to the program. By separating the data from the program, the change of input source from cards to tape or disk would involve changes to the DD statement only; no SAS changes would be required. There are also other reasons for keeping data and program in separate datasets.

SYSIN DD Statement: This JCL statement defines (preceeds) the dataset containing the SAS program.

The LOG and PRINT Files. SAS programs typically produce two printed output datasets called the LOG and PRINT files, respectively. The program listing, NOTEs from the SAS supervisor, and SAS error messages are printed in the LOG. Following the LOG, the PRINT file contains printed output produce by any PROCs which might be executed. The data input program may place output in either file.

The TITLE and TITLE2 statements in the example specify titles to be printed by SAS at the top of each page of the PRINT file (not the LOG file). TITLE statement are discussed in SAS79 on page 18. The reader should study that description of this point. Please see the printed output from the example (Figure 2.4) to observe the effect of the TITLE statements. In addition to labelling the printed output the TITLE statements form an important part of the documentation of the data processing activities. COMMENT statements within the program also serve this documentation purpose.

2.2 DATA STATEMENT

The DATA statement in the example, which is typical of data input phase DATA statements, performs the following functions:

1. It is the first statement of a DATA step.
2. It names the output SAS database ("PERM") and the output SAS dataset ("PFD79")
3. It provides certain other information about the output dataset, in this case a dataset label, which contains a description of the dataset longer than can be squeezed into an 8-character name.

In their most general form DATA statements may be quite complicated. The most commonly used form, illustrated in the example, should be imitated by the reader until more advanced forms have been discussed.

2.3. The INFILE statement

The INFILE statment specifies to the SAS supervisor the ddname of the OS dataset containing the input data. (Note: including the data in the program following a "CARDS"; statement - which is not recommended- is an alternative to the use of the INFILE). INFILE statements may be quite complicated in some cases and SAS79, a reference manual, describes all the possible options. Reading about INFILE in SAS79 now is not recommended: wait until more advanced topics have been covered. For the present, simply imitate the usage shown in the example, but feel free to create your own ddnames (to replace "TARJETAS").

2.4 Points of Programming Style:

The rules for creating SAS names, such as names of variables, names of datasets, names of databases, ddnames, etc., are described on pages 7-8 of SAS79. The reader should review that description at this point.

Please do not use names which begin and end with underscores, such as _ALL_, _N_, etc. SAS has a number of special names using this format. If you use this format you might accidentally use a "magic" word and create some debugging problems.

Indenting. Note that all the statements in the example which follow the DATA statement are indented. This type of program formatting is not required by SAS but is highly recommended, as are other types of indention illustrated in the Example. In this case, the indention shows at a glance which statements are included in each SAS step of the program. Such indention therefore partially documents the structure of the program. Proper program formatting is also a very useful debugging tool.

2.5 The LENGTH Statement

The LENGTH statement is a very important declaration statement which specifies to the SAS compiler:

1. The type (numeric or character) of each variable listed in the statement, and
2. The length of each listed variable, that is the number of bytes to be used for storing a value of the variable.

A variable is declared to be character if a '$' follows its name in the LENGTH statement; any variable name in a LENGTH not followed by a '$' is declared to be numeric.

Each variable has an associated length, in bytes. Numeric variables must have at least two bytes; their maximum length is 8 bytes (approximately 15 decimal digits plus a 1-byte exponent). Character variables may be any length from 1 byte to 200 bytes.

Default Length and Type of Variables. The SAS compiler determines both the byte and length of a variable at the first occurrence of the name of the variable in the program. If, at that point, there is no explicit information about the type and/or length, SAS will use numeric as the default type and/or 8 bytes as the default length. (See the description of the LENGTH statement in SAS79, page 54).

In the example only four variables are explicitly declared as to type and length in the LENGTH statement.

SAS will make DATEBORN and SURVDATE numeric when it encounters them in the FORMAT statement because a numeric format (yymmdd8) is used. The lengths will be the default, 8 bytes.

SAS will make CARDTYPE a character variable when it is encountered in the INPUT statement because of the '$' following the name. SAS will determine CARDTYPE's length to be 5 bytes because that is the number of bytes in the input field.

IDNUM will be declared numeric when its name is encountered in the INPUT statement because there is no '$' or character format following the name. Regardless of the input field with, numeric variables have a default length of 8 bytes. Thus, IDNUM will have a lerght of 8 bytes.

The variable AGE will default to an 8-byte numeric variable when the compiler encounters its name in the LABEL statement.

The type and length of a variable are very important. Using too-long

lengths for variables in large datasets wastes storage space and can increase processing time. And, although SAS permits a programmer to mix character and numerical variables in the statement, SAS uses very complicated operational rules in such cases and the complications frequently lead to difficult debugging problems.

We recommend that all variables in a data input program be declared in LENGTH statements (and/or RETAIN statements, discussed later). In the example we omitted variables from the LENGTH in order to raise the issues discussed above.

2.6 The LABEL Statement

Although the SAS names for variables are mnemonic and somewhat descriptive there are limits to the amount of information one can cram into an 8-bytes name. SAS provides for storing up to 40 bytes of additional information about each variable in a "variable label". Variable labels are specified in one or more LABEL statements. (See SAS79, page 112 for detailed specifications.) The information in a label needs be specified only once. The information is stored in the dataset along with the name of the variable, its length, etc. This information is 'carried along' through any future processing steps. Many of the analysis PROCs print the label along with the name of the variable, when the label is present.

Preparing a descriptive label for every non-obvious variable is good programming practice. Only an amateur would fail to take advantage of this documentation feature. We omitted labels from some variables in the example to make this point.

2.7 DATE Variables

One of the beautiful features of SAS79 is its ability to perform date computations automatically. This feature is extremely useful in processing research data, as in the example, where AGE is the difference between two dates.

In SAS79, a "DATE Variable" is simply a numeric variable whose value represents the number of days (an integer) from 01 January 1960 to a given date. Since a DATE variable is a numeric variable, special arithmetic is not necessary. But SAS provides special functions for converting from external representation of dates (e.g., 03Feb80) to the correct internal representation, and back again.

The example has two DATE variables, DATEBORN and SURVDATE which are input (see the INPUT statement) from 6-byte fields in the format yymmdd. On input, SAS converts the 6 digits (e.g.800203 for 03Feb80) to a value representing the number of days from 01Jan60 to the data base (e.g., to 03Feb80). Leap years and other calendar anomalies are correctly accounted for.

The difference between two DATE variables e.g., SURVDATE-DATEBORN is not a DATE value because it does not represent the number of days from 01JAN60 to a specific date.

The difference of two DATE values is the number of days between the dates. Thus, in the example, age in years is computed as AGE=(SURVDATE-DATEBORN)/365.25. The extra 0.25 in the denominator represents the fact that a year is (to 5 digits) actually 365.25 days long, creating the need for leap years. We will present additional discussion of date variables in a later section.

2.8 The FORMAT Statement

Since DATE variables are simply numeric variables with special interpretation, SAS will use a standard numeric default format to print them unless directed otherwise. While it may be interesting to see a date represented as the number of days since 01JAN60, humans find other date formats (e.g., 03FEB80 800203, 80-02-03, and February 3, 1980) much easier to work with.

The FORMAT statement allows the SAS programmer to associate a specific printing format with a variable. Then, when values of the variable are printed

later, SAS will use this previously-specified format for converting from internal to external form. (The programmer may override the format at a later time). The format information is stored in the dataset along with the name of the variable, its length, its label, etc. A format may be associated with either type of SAS variable.

Formats are a topic to be studied in depth at a later point. Here it is sufficient to say that one uses the FORMAT statement for this purpose and that the example illustrates a straightforward usage. In the examples, the output format specified for both DATEBORN and SURVDATE is of the form yy-mm-dd, that is, 8 bytes including the hyphens.

2.9 The INPUT Statement

The INPUT statement is the heart of a Data Input Phase program. The INPUT statement specifies to SAS the name of each variable, the location in the input record of the data for the variable, and special information (the format) for converting from external format to internal format.

This section contains only elementary information on INPUT statements and formats. Another section is devoted to more extensive discussion of the topic.

The INPUT statement in Figure 2.2 is typical of INPUT statements for data with one fixed-format card per observation. The keyword INPUT is followed by one 'specification' for each data field. In the example the specifications are placed on successive lines so that the statement has the apperance of a table, defining the input format of the data. SAS does not require that the INPUT statement be arranged in this manner, but itis good programming practice to do so.

The first field specification, "CARDTYPE $ 1-5", has three parts, the variable name (CARDTYPE), the "$" symbol, which means that the field is a character string (not numeric) and the specification of the card columns for the field, "1-5" denoting columns 1 through 5. (The "-" symbol is used

here as a hyphen rather than a minus sign). For each card (and observation) this specification will cause the character string in columns 1-5 of the data card to be moved into the memory locations associated with CARDTYPE. In addition, leading blanks, as "␣ABCD" (where '␣' denotes the blank character), will be deleted by shifting non-blank characters to the left and inserting blanks on the right. That is, the input character string '␣ABCD' would be stored in the SAS dataset as 'ABCD␣'. This process is called 'left justification' of the character field. (This process can be overridden by using a $CHARw. format, discussed on pages 36-37 of SAS79)

The second specification, "IDNUM6-9", is similar to the first except that the omission of the "$" indicates that the field and the variable are numeric. SAS will convert the value, punched in columns 6-9 to internal numerical format. Leading and/or trailing blanks will be ignored. (This is different from Fortran, for example, which treats trailing blanks as zeroes.) If a decimal is punched in the field, the number will be converted into the correct internal representation, including the fractional part. "E-notation" is also converted properly: '12E4' would be converted to the internal equivalent of 120000, for example.

The fields for NAME, SEX, HEIGHT, and WEIGHT will be processed in a manner similar to the processing of CARDTYPE and IDNUM.

2.10 Formats in INPUT Statements

Data fields other than straightforward numeric and character fields require special additional information called 'format' for proper conversion to internal format. SAS provides a staggering variety of formats for such special cases. (SAS79, chapter 5, pages 29-44).

The example INPUT statement in Figure 2.2 contains an example of the use of specialized formats. The variables DATEBORN and SURVDATE are SAS date variable (discussed above in section 2.7). In a data card, each of these variables occupies 6 columns in the format yymmdd, where yy represents the last two digits of the year, mm represents the month number (01 for January, 02 for

February, etc.) and dd represents day number within month. An example would be '800203' for 03Feb80. Unless instructed otherwise, SAS would treat the data as numeric, having the value 800203. By directing SAS to use a date format (YYMMDD6. in this case; see figure 2.2), the programmer instructs SAS to convert the 6-digit number into a date variable representing the number of days since 01JAN60. In the example, the value '800203' read from the data card would be converted to 7338, the number of days from 01JAN60 to 02FEB80.

2.11 PROC PRINT

The statement "PROC PRINT;" in the example (Figure 2.2) executes the SAS PRINT procedure. Notice that this SAS step has two statements, "PROC PRINT"; and "TITLE3..". The TITLE3 statement adds a third line to the title information generated by TITLE statements in the first step.

The PRINT procedure produces a nicely formatted printout of the contents of a SAS dataset. The name of the input dataset may be specified. If it is omitted, the contents of the most recently created dataset (in this job) will be printed.

The PRINT procedure is very simple to use; it is described on page 353 of SAS79. The reader should read the SAS79 description of PROC PRINT at this point.

Note that the example contains another execution of PROC PRINT after the data are sorted (PROC SORT). The second execution of PRINT uses the ID statement, which is described in SAS79, page 353.

2.12 Sorting: First Look at PROC SORT

Sorting a SAS dataset is delightfully easy. The example (Figure 2.2) illustrates a simple execution of PROC SORT.

Generally, one would specify an input SAS dataset and a different output SAS dataset for the SORT procedure, in the following form:

```
PROC SORT DATA = input SAS dataset
          OUT = output SAS dataset;
BY list;
```

Note the use of the parameter DATA to specify an input dataset. This is in contrast to the use of the statement DATA to specify: (1) the beginning of a DATA step and (2) the name of an output dataset. The parameter DATA (in a PROC statement) is completely different from the DATA statement!

If one omits the DATA parameter from the PROC SORT statement (as in the example), SAS will use for input the most recently created SAS dataset in the current job.

If one omits the output dataset specification (OUT=) in the PROC SORT statement, SAS will, in effect, make the output dataset name the same as the input dataset name. If something causes PROC SORT to fail, or terminate before completing output dataset, the input dataset will be unaffected.

A PROC SORT step must always include a BY statement, which specifies the variables whose values are to be used to determine the ordering of the observations. In the example, the data are sorted on the character variable NAME, this orders the observations alphabetically, by values stored in NAME.

The reader should now study the SAS 79 discussion of the SORT procedure, pages 373-375, except for the section "Sorting Large Datasets".

2.13 First Look at PROC CONTENTS

PROC CONTENTS is a SAS procedure which prints out information about a SAS database and the SAS datasets contained in the database. This procedure is an important tool, along with PROC DATASETS and PROC DELETE, for the

documentation and "management" of SAS databases. Although PROC CONTENTS was not included in the example, it is included in the assignment. The reader should read the SAS79 section on PROC CONTENTS at this point.

2.14 SUMMARY

We have presented, by way of an example, a "typical" Data Input Phase program. The only major omission is that the example program does not check for data errors. The SAS tools (statements) used for error detection are discussed in subsequent sections.

Remember that the principal objective of a SAS Data Input Phase program is to get the data into a SAS dataset. SAS datasets are easier to process, modify, and manipulate that OS datasets.

If you are a newcomer to SAS and computer processing, much of the material in this section may have been unfamiliar and some probably did not make sense. Beginner or not, it is important to do the exercise in the next section which may raise important questions to be directed to your instructor or local SAS expert. Beginners should not be troubled by their present state of confusion: do the exercise, re-read this chapter, and then re-do the exercise. As noted in the introduction, learning data management is an iterative process.

2.15 Exercises/Assignment:

1. Read each section of SAS79 referenced in Section 2. Note that in some cases reading an entire chapter in SAS79 is not recommended (because the chapter contains descriptions of advanced features not yet covered)

2. Create an OS dataset on disk to be used for later examples. You will probably need assistance from the instructor or local SAS expert.

3. Obtain a copy of the data described in the example. (The format is in figure 2.3) Make a SAS program to input and sort the data and store the results in your SAS database. (Feel free to copy the program in Figure 2.2. You might wish to add labels for the 'unlabelled' variables).

Add a PROC CONTENTS step at the end of the program to print information about all the datasets in you database.

Incidentally you should expect to make errors. Examine the LOG section of your output carefully (especially the "NOTES") to determine if the program has run properly.

3. Numerical Operators Expressions, Statements, and Functions

One of the very nice features of SAS is the ease with which new variables can be created from old ones. An almost unlimited variety of numerical transformations and combinations can be made with numerical expressions, statements, and functions.

3.1 Numerical Operators, Assignment Statements and Expressions

A typical numerical statement is in the format.

variable = expression;

where variable is the name of numerical variable and expression is a SAS numerical expression.

SAS numerical expressions are very similar to numerical expressions in other computing languages such as Fortran and PL/1. We assume the reader has some familiarity with such expressions in other languages.

The SAS numerical operators (defined in SAS79, Appendix 5) are:

Operator	Name	SAS Code	Math
**	Exponentiation	A**B	A^B
*	Multiplication	C*D	$C \cdot D$
/	Division	E/F	$E \div F$
+	Addition	G+H	G+H
-	Subtraction	I-J	I-J
><	Minimum	K><L	Min (K,L)
<>	Maximum	M<>N	Max (M,N)

One combines constants, variables, and operators to form expressions. In the preceeding example we had the statement.

```
AGE = (SURVDATE-DATEBORN)/365.25;
```

which converts age, in days, to age, in years.

Numeric constants, such as 365.25, are generally coded just as they appear. For special forms (E-notation, hexadecimal), see Appendix 5 of SAS79.

"Operator Precedence' One can use parentheses to define the order of several operations, as in the expression above. In some cases one also uses the fact that, if parentheses do not define the order of execution of operations,

(a) Operators with high precedence are executed before operators with low precedence.

(b) Operators with equal precedence are executed in left-to-right order, except in the first group. (See Appendix 5. If in doubt, use parentheses)

The operator precedence table is:

Highest:	**, prefix -, prefix+, >< , <>
	*, /
	comparison operators (<, <=, etc.
	AND
Lowest:	OR

Thus, for example, one could code

$$a = \frac{c}{d} + \frac{e}{f+g} + \frac{h + i}{J}$$

As

```
A**C/D+E/(F+G) + (H+I)/J
```

Here, both sets of parentheses are required. Similary

$x = y^2 + z^2$ could be coded

as
```
x  SQRT (Y*Y + Z * Z)
```
or
```
x  SQRT (Y** 2 + Z ** 2)
```
or
```
x  (· ** 2 + Z ** 2)  ** 0.5
```

In this example, the first code is preferable because Y ** 2 is evaluated as Exp (2* Log (Y)). Generally, A ** B is evaluated as Exp (B * Log (A)). This expression fails and creates error messages if A<0, or Y<0, even though Y^2 is perfectly well defined for Y<0. Thus, low order integer powers, as Y^2, Y^3, Y^4, are best coded as repeat multiplication: Y*Y, Y*Y*Y, etc.

3.2 Arithmetic Functions

SAS provides a wide variety of numerical functions, described in Appendix 1 of SAS79. The Arithmetic functions are:

ABS	-	Absolute value
CEIL	-	Smallest integer ≥ argument
FLOOR	-	Largest integer ≤ argument

INT	-	Truncation of argument to integer
MOD	-	Mod (a, b) = a mod b
SQRT	-	Square root of argument
ROUND	-	Rounds to nearest integer
SIGN	-	= ± 1., depending sign of argument

(SAS also lists MIN and MAX among the arithmetic functions,) but these are really statistical functions.)

Trigonometric and Hyperbolic Functions. SAS provides all the usual trigonometric and hyperbolic functions: SIN, COS, TAN, ARCOS (arc cosine), ARSIN (arc sine), ARTAN, COSH, SINH, TANH.

Math Functions. SAS provides a useful collection of math and math-stat functions: LOG, LOG10 and LOG2 logarithm to base e, 10, and 2, respectively.

GAMMA, LGAMMA	Gamma function and log gamma function, respectively.
GAMINV	Inverse gamma function
EXP	Exponentiation: e^x.
PROBNORM, PROBCHI, PROBT, PROBF, PROBGAM	Probabilities computed from the indicated distribution (normal, chi-square, t, F, Gamma)

Random Number Generators - SAS provides UNIFORM and NORMAL to generate random numbers from the standard uniform (0, 1) and standard normal, N(0,1), distributions.

3.3 Statistical Functions. SAS provides 15 functions for computing statistics from the arguments. These functions are distinctive in that:

- The number of arguments is not fixed - any number of arguments may be used.

  For example:

  X = MEAN (Y1, Y2, Y3); Y=MEAN (Y1, Y2, Y3, Y4);

- Any argument which has a missing value is ignored in the computations

(except the functions N, NMISS).

- A special form of argument can be used for long lists of variables.

The statistical functions are listed and defined on page 444, of SAS79. They are: N, NMISS, SUM, MEAN, MIN, MAX, RANGE, etc.

In "standard use", the arguments are separated by commas:

```
X = MIN (3, X2, X3, X7);
```

For these statistical functions only, lists of variables whose names end in numbers can be written in the form

```
Y = MAX (OF X1 - X100)
```

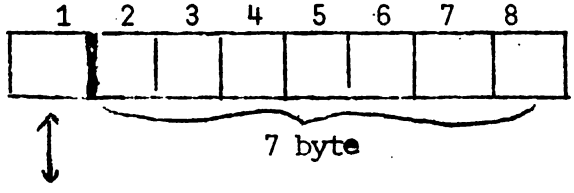
which has the same effect as writing out all of X1, X2,... X100, separated by commas.

As noted above, these statistical functions ignore missing values in the computations, except for the functions N and NMISS. N counts the number of arguments with nonmissing values and N counts the number of arguments with missing values.

3.4 Missing Values:

Generally, on input, if a data field for a numeric variable is blank, SAS will store a special "missing value" for the variable in that observation. Normally a numeric value is stored in an 8-byte hexadecimal floating point format as follows:

BYTE:



Sign (first bit) and base-16 exponent (7 bites)

If the data value is zero, SAS stores a zero mantissa, a + sign, and an exponent value of 0 (for 16°). If the data value is missing, SAS stores a zero mantissa and the character "b" (a blank character, hex 40) in the sign/exponent byte. In later processing, before performing arithmetic operations, SAS checks each numeric value to see if it is a "missing value". If so, the result of the operation is a missing value.

A programmer can code missing values with the character ".", in an SASnumeric expression for example: X = .;

= will place a missing value in X.

SAS actually permits 28 types of missing values, which may be coded as : ".", "._", ".A",".B", ".C", .., ".Z". For ".", SAS stores a "b" (blank character) in the first byte; for "._" SAS stores an "_" character; for ".A" through ".Z" SAS stores the character "A", ..., "Z" in the first byte.

The missing value "." is called the "simple missing value". It is normally placed in a numeric variable when the input field is blank.

The other missing values are called "special missing values". One can use these in program statements, as X = .A; these can also be generated by SAS by use of the MISSING statement.

All of this is described nicely in chapter 6, of SAS79 Missing Values, which should be read at this point.

The RETAIN Statement:

In "normal" operation, before beginning initial processing, and after outputting an observation to a SAS dataset, SAS sets all numeric variables to simple missing values (".") and all character variables to blanks.

In this mode of operation, the programmer cannot access any information from the preceeding observation. For example, suppose one wishes to count the number of observations. A programmer experienced in FORTRAN or PL/I might

try the following program:

```
N=0;
DATA  name;
   ...
   INPUT list;
   N= N + 1;
```

This program will not work. SAS will not accept the "N=0"; statement outside a DATA step. Inside the DATA step, the result of the "N=N + 1 ; " statement (assuming N is not in the INPUT list) would be to set N to a missing value, because N would be set to a missing value before each execution of the statements in the program.

The RETAIN statement allows a programmer to specify variables which will <u>not</u> be set to missing or blank values before execution of the statements in the step. The programmer can also specify an initial value (constant) <u>other</u> than missing (".") for a variable with the RETAIN statement.

The example above could be accomplished by the following program:

```
DATA name;
      RETAIN N (0);
      INPUT list;
      N = N+1;
      (other statements)
```

Here, N is specified as a variable whose value will not be changed after SAS outputs each observation. Moreover, N is initially set to zero. Thus, the "N= N+1;" statement following the INPUT statement effectively counts the

number of times the INPUT is executed.

The RETAIN statement is described on pages 107-109 of SAS79, which should be read at this point.

The reader should be aware that SAS provides other means of accomplishing the counting of records (as above) and similar tasks. ("Sum" statements, p. 103, may be used, for example, or the automatic variable _N_, which serves a similar purpose). Sections of SAS79 describing these features should be read when the need to use them arises.

4. Character Manipulation

SAS offers somewhat limited, but very useful facilities for storing and manipulating character variables, i. e., variables whose values are strings of EBCDIC character rather than numbers to be used for computation.

4.1. Character Variables

SAS has two types of variables: (1) numeric variables, whose values are stored in a special internal format suitable for numerical computation, and (2) character variables, whose values have the same internal and external format (EBCDIC). Character values, or "strings", cannot be used directly for computation. If the character string contains a number in EBCDIC format, the string may be converted to internal numeric format for use in numerical computations. Or, the value may be maintained in EBCDIC format and manipulated as a character string.

Each character in a character string requires one byte for storage. A blank character ('b') is simply one of the available characters and therefore also requires one byte for storage. (We shall use 'b' to denote the blank character).

A SAS character variable has a fixed length, which is the number of bytes available for storing values of the variable.

Determining the Type and Length of Character Variables:

In a DATA ... INPUT step the SAS compiler determines the type of a variable (character or numeric) and the length (number of bytes allocated for string values) from the first appearance of the variable's name in the

SAS program. If a variable is not specified to be a character variable, SAS will assume the variable is numeric.

SAS will try to determine the intended length of a character variable from the available information. If the first appearance of the character variable is in an INPUT statement, SAS will determine the length from this statement. For example, in the previous example creating the PFD79 dataset, the variable name CARDTYPE first appears in the statement.

```
INPUT CARDTYPE  $  1-2  etc.
```

The "$" indicates that CARDTYPE is to be a character variable. SAS determines from the specified input columns (1-2) that the variable is two bytes long.

SAS may also determine the type and length of a character variable from its first appearance in an assignment statement. If the INPUT statement above were followed by

```
A  =  CARDTYPE;
```

Then SAS would define A as a character variable with the same length as CARDTYPE.

The programmer can specify the type and length of character (and numeric) variables in LENGTH statements as described in a preceeding section. A LENGTH statement is a declaration, or definitions and is not "executable". Once the type and length of a variable are established they cannot easily be changed. The LENGTH statement is described on page 54 of the SAS79 manual and is illustrated in the PFD79 example.

The LENGTH statement must follow the DATA statement and preceed any other statements containing the names of the variables in the LENGTH statement.

It is good programming practice to explicitly specify the type and length of character variables in LENGTH statements.

4.2. "Missing Values" for characters Variables. Since character variables cannot participate directly in numeric computations, there is no "missing value" problem for character variables. Any valid EBCDIC character in the input data may be input to a character variable. A blank input field results in the character variable containing blank characters.

4.3. Conversion Between Character and Numeric Values:

The SAS compiler will attempt an automatic conversion of a character value to a numeric value, or vice versa, when character and numeric values are mixed in the same statement. For example, if X is a numeric variable and C is a character variable of length 8, the statement,

```
C = X;
```

would be compiled so that the numeric value in X would be converted to a 12-byte EBCDIC character string (using the BEST12. format), the rightmost 4 characters would be deleted, and the leftmost 8 characters would be stored in C. (See page 11 of the SAS79 manual for more details). This may or may not produce the result the programmer wants.

The statement,

```
X = C;
```

would be compiled so that SAS would attempt to convert the 8 EBCDIC characters in C to internal numeric format. If the conversion is successful, the resulting number is stored in X. If the conversion is not successful (e. g., if C = 'ABCDEFGH'), a simple missing value (.) is placed in X.

SAS will attempt conversions wherever needed. For example, using C above, the statement "X = SQRT (C);" would have essentially the same result as the two statements,

```
TEMP = C;
X = SQRT (TEMP);
```

where TEMP is a numeric variable.

The two character functions PUT and INPUT (SAS79, page 439) are available to the programmer to perform explicit transformations between numeric and character values. One can reliably obtain the required results using these functions. Letting SAS perform the conversions implicitly is poor programming practice.

4.4 Comparison of Character Values

The SAS comparison operators (<, <=, =, =>, >, >=) can be used to compare character values. The result of such a comparison is always a "logical" value (i.e., a numerical value with 1 representing "TRUE" and a 0 representing "FALSE").
(Comparison of character values is discussed in detail in Appendix 5 of SAS79, pages 458-459).

If two character values of different lengths are to be compared, SAS will move the shorter value to a temporary storage location, add blanks to the right until the temporary is the same length as the other value, and then make the comparison between two strings of equal length.

For purposes of determining the ordering of characters (e.g., is '%'less than "$" ?) SAS uses the standard IBM 360/370 "collating sequence.
The order of all the common characters is shown on page 458 of SAS79. In this sequence, the groups of "printable" characters are:

(blank) < special characters < letters < numerals

The special characters are such as: .< (+ etc.

The letters are A < B < .... < Z and the numerals (EBCDIC codes for digits) are 0 < 1 < 2 < ... < 9.

4.5 Concatenation

The only SAS operator which "combines" two character values to produce another character value is the concatenation operator, ||, described on page 459 of SAS79. (The reader should study the description in SAS79 before proceeding).

Concatenating names may be a bit tricky because the || operator trims (deletes) blanks before concatenating. For example, the following code,

```
LASTNAME = 'PAEZ ';
FIRSTN   = 'GILBERTO ';
NAME     = LASTNAME || FIRSTN;
```

would produce the same result as

```
NAME     = 'PAEZGILBERTO';
```

One way to solve the problem is to insert a blank or comma:

```
NAME     =  LASTNAME || ' ' || FIRSTN;
NAME     =  LASTNAME || ', ' || FIRSTN;
```

4.6 Character Functions

The functions provided by SAS for manipulating character values are:

SUBSTR	to extract a substring;
INPUT	to explicitly transform an EBCDIC string into internal format, using a SAS format;
PUT	to explicitly transform an internal value (numeric or character string format, using a SAS format).
LENGTH	to permit the program to determine the length of a character string, omitting blanks on the right;
REVERSE	to reverse the order of characters in a string.

The technical details of these functions are given on page 439 of SAS79.

4.7 Character Formats

SAS provides for character formats, for use in INPUT, PUT, and related statements (and the INPUT and PUT functions) for transforming characters from external storage to or from internal storage. The formats, $w., $CHARw., $HEXw., and $VARYINGw., are described on pages 36-37 of SAS79. There is very little difference between internal and external storage of

character strings. The principal difference is in the handling of leading blanks and the length.

If an input string is too long for the character variable, the string is truncated on the right, to fit. (The rightmost characters are deleted until the string is short enough to fit). If an input string is shorter than the character variable, the string is padded on the right with blanks. (Blanks are added on the right until the string is the same length as the variable.)

The "$w." format causes leading blanks to be deleted from a string on input; the first character of the stored result will be non-black unless the entire value is blank. The "$CHARw." format does not delete leading blanks on input.

The "$VARYING." format is very useful for "free format" input character data. The use of this format will be illustrated subsequently.

5. Logic Statements: IF, THEN, ELSE, GO TO, etc.

5.1 Logical (Boolean) values, variables and expressions

Logical (Boolean) variables take only two possible values, TRUE and FALSE. For convenience, SAS (and many other programming languages) code these values numerically as 1 for TRUE, 0 for FALSE. Thus, in SAS, a "logical variable" is really just a numeric variable; the programmer forces the variable to be Boolean by restricting it to the values 0 and 1.

Logical (Boolean) expressions are built up from the Boolean operators, AND (&), OR (|), and NOT (^), from Boolean values and variables, and from comparisons. (See SAS79, Appendix 5, for more details).

Comparisons are expressions of the form: display (variable (or value)) (comparison operator) (variable (or value), as for example,

```
(A   B)  AND   NAME = 'JONES'
```

In SAS, comparisons always produce a result of 0 or 1.

Comparisons may be combined by Boolean operators to produce Boolean expressions as, for example,

```
(A  B)  AND  (NAME = 'JONES')
```

Parentheses are used here to make the order of evaluation explicit. The value of a comparison or Boolean expression may be sorted in a numeric variable:

```
X = (A  B)  &  (NAME = 'JONES');
```

One can also perform numeric computations on Boolean values:

```
NUM = (X1  5) + (X2  5) + (X3  5) + X4  5);
```

Here, NUM is the number of variables (among X1, X2, X3, X4) which exceed 5.

The principal use of Boolean expressions is in IF statements, discussed below.

5.2 Statement Labels and GO TO

Within each execution of a SAS step, the SAS statements are "normally" executed in the order in which they appear in the program. The programmer uses statement labels and the GO TO statement to execute the statements in a different order.

A statement label is a unique SAS name at the beginning of a statement, followed by a colon as, for example:

```
TOP:   X= X+1;
```

The label, TOP, identifies the statement.

The "GO TO label". Statement transfers control to the statement indicated by label. This may be a "forward" or "backward" jump as is illustrated by the program segment:

```
STEPA:   A = B + C
         GO TO STEP2;
STEP1:   A = D + E;
         GO TO STEP A;
STEP2:   Y = SQRT  (Z);
            ...
```

Here, the first GO TO jumps forward, the second jumps back. More details on statement labels and GO TO are given on pages 113-114 of SAS79.

5.3 DO-END Groups.

Sometimes, as in IF statements discussed below, it is convenient to treat a group of statements as a block, almost as one statement. Such a block of statements as specified with an initial "DO;" statement and a terminal "END;" statement as, for example:

```
TOP:  DO;
        X = X+1 ;
        Z = SQRT (X);
        T = X* Z;
        END;
```

(Indenting the statements in a "DO group" is not required by SAS but is good programming practice). SAS treats all the statements in a DO-END group as a single block, as will be seen in the next section.

5.4 IF-THEN and IF-THEN-ELSE

The most important "control" statements in SAS are the IF statements. A complete explanation of IF statements requires several chapters of a book on programming, much more than the few paragraphs available here. (A very brief description of IF-THEN-ELSE statements is found on page 113 of SAS79. The reader should read that description at this point.)

The first principal use of IF statements in RDM is for detecting data errors, such as checking if a variable's values are in a reasonable range:

```
ERR = 0;
IF  AGE < = 0   OR AGE> = 80
    THEN  DO;
      PUT /'AGE OUT OF RANGE';
      ERR = 1;
      END;
```

or if a variable has a valid value:

```
IF SEX ^= 'M' &  SEX  NE 'F'
   THEN  DO;
         PUT /'INVALID VALUE FOR SEX';
         ERR  =  1;
   END;
```

The other primary use is to selectively execute certain statements, as:

```
IF  ERR  THEN  LIST;
```

This statement, following the ones above, would print in the LOG the values of all the variables when an error had been discovered (when ERR = 1).

Note that the types of statements which may follow THEN or ELSE is limited (SAS79, page 113); in particular,and IF may not follow THEN or ELSE. A DO can be used to solve the problem:

```
ERR = 0;
IF SEX = 'M'
THEN DO;
  IF HEIGHT > 210  THEN ERR = 1;
END;
ELSE DO;
  IF HEIGHT > 190 THEN ERR = 1;
END;
```

Note the code above could be abbreviated by using ERR as a Boolean variable:

```
ERR = 0
IF SEX = 'M'  THEN ERR = ERR or HEIGHT>210;
              ELSE ERR = ERR or HEIGHT>190;
```

5.5 ERROR, STOP and ABORT

Sometimes error conditions arise and the programmer wants his program to signal the condition and/or terminate processing. The ERROR, STOP and ABORT statements provide three levels of action for such conditions. These statements are described on pages 106-107 of SAS79.

Briefly, "ERROR;" or "ERROR message;" causes the system error indicator for the current observation, _ERROR_, to be set to '1'. If a "message" is present, it is printed. Upon completion of processing of the observation, if _ERROR_ = 1, the values of all the variables are printed in the LOG (as if LIST were executed).

Execution of STOP terminates execution of the current SAS step. SAS will go to the next SAS step (DATA or PROC) and attempt to continue.

Execution of ABORT terminates SAS processing; control is returned to the operating system. ABORT also permits the user to ABEND the job with a user completion code. (See SAS79 page 107 for details).

II. 6. Arrays and Loops

6.1. Arrays

In SAS79, an array is a collection of variables to be processed together in DATA steps. The advantage of an array is that a single name (the name of the array) can be used to represent any variable in the array.

All the variables in an array must be the same type (character or numeric). The lengths of the variables may vary.

An array is defined in the ARRAY statement (a declaration, a non-executable statement), which has the general form (SAS79, pages 12-13):

ARRAY array name (index name) array elements;

The example in SAS79 is

ARRAY Q (I) Q1-Q20;

The name of the array is Q (a unique name within the DATA step). The array index is I. The variables Q1, Q2, ..., Q20 are elements of the array.

Whenever the array index, I, has a value, $1 <= I <= 20$, the variable name Q refers to the I-th element of the array. For example, when I=1, Q is the same as Q1; when I=2, Q is the same as 02, etc.

Implementation:

When the SAS compiler encounters an ARRAY statement, it prepares a list of the variables in the array and marks the array name as a special type of variable. When the array name is encountered in a subsequent statement the compiler creates special code. When this code is executed, it determines the value of the index variable ("I" in the example), finds the address of the indexed element (variable), and uses this address as the effective address. (This is called a "pointer implementation" because the array index is really a pointer variable.)

The implementation strongly resembles the use of PL/I pointer variables and is very different from the implementation of arrays in FORTRAN or PL/I.

Arrays have a wide variety of uses in RDM programming, some of which will be illustrated in a subsequent section.

6.2 Variable Lists

SAS79 programs typically contain a number of lists of names of variables, or "variable lists". SAS provides several methods of abbreviating certain types of variable lists. (See SAS79, page 9).

The most elementary form applies to "numbered names", variable names of the form "aaaaannn", where "a" represents a letter (or underscore) and "nnn" represents a number with no leading zeroes. Examples (from SAS79) include

Q2 Q3 Q4 Q5

SEX1 SEX2 SEX3

etc. A list such as the one above can be abbreviated in the form

aaaaannn - aaaaammm

where "mmm" is a number larger than nnn. The lists above become

Q2 - Q5 and SEX1 - SEX3

All of the implied intermediate variables must exist, e.g., in Q2 - Q5, Q3 and Q4 must exist.

SAS also permits specification of "ranges of variables", but this technique requires exact knowledge of the order in which SAS is storing variables in an observation. This technique is not recommended for professional-quality programs because subsequent modification of one part of a program, which alters the order in which variables are stored, will probably introduce errors into the statement using a range of variables specification.

The abbreviation _NUMERIC_ as a variable list is translated to a list including all the numeric variables defined at that spot in the program. The abbreviations _CHARACTER_, and _ALL_ produce similar results for character variables and all (numeric and character) variables defined at that spot in the program.

Be careful with variable lists. If in doubt, make an exact list containing each desired name. If a list is used repeatedly, a MACRO can save work.

Variable lists have an obvious application in ARRAY statements.

6.3 DO Loops.

DO loops are among the most powerful of all control statements. Loops, together with arrays, provide programming capabilities which are extremely difficult to program with other combinations of statements.

(DO statements are described on pages 115-116 of SAS79).

Iterative DO. The iterative DO statement has one of two general forms,

```
DO index = start TO stop;
DO index = start TO stop BY increment;
```

For example, consider the following part of a program, in which X1, X2, ..., X100 are assumed to have values (by INPUT, or otherwise) and it is desired to count the number of Xi which have abosolute value greater than 0.5.

```
ARRAY  X  (I)  X1 - X100;
NUM = 0;
DO  I = 1 TO 100;
IF  ABS  (X) > 0.5  THEN NUM = NUM + 1;
END;
```

The IF statement will be executed 100 times; on each execution X will represent a successive variable, X1, X2, ..., X100.

The SAS79 DO is very similar to the PL/I DO; the reader should examine the documentation in SAS79 (pages 115-116).

DO OVER. The array statement is matched with the iterative DO in the DO OVER statement, described on page 116 of SAS79. This form of the DO is especially useful for general programs or "subprograms" for RDM problems.

II. 7. Macros and Subroutines

SAS provides Macro statements and subroutine capabilities to let the programmer re-use groups of program statements.

7.1 Macros

The purpose of a MACRO statement is to modify the actual SAS program by repeatedly inserting program text. A typical use occurs when a programmer

has a long list of variables which must be used in several places in a program and which cannot be abbreviated by one of the techniques discussed earlier. We will illustrate with a short list of variables to reduce the size of the example shown in Figure 2.6.

Figure 2.6 Example of the Use of MACRO statements.

```
//SYSIN   DD *
  MACRO IDLIST PROVINCE CANTON DISTRICT IDNUM  %
  MACRO VARLIST IDLIST FIELDNUM
     CROP ALTITUDE SOILTYPE
     RAIN_JAN RAIN_FEB RAIN_MAR  %
  DATA DATABASE.FARM;
    {various DATA Step Statements}
  PROC PRINT,
     ID IDLIST;
     VAR VARLIST;
     TITLE INPUT DATA IN ORIGINAL ORDER;
  PROC SORT; By IDLIST;
  PROC PRINT;
     ID IDLIST;
     VAR VARLIST
     TITLE INPUT DATA SORTED BY IDLIST;
```

Figure 2.7. Effect of the MACRO statements in Figure 2.6.

The First PROC PRINT Step After MACRO Text Substitution Becomes as Follows:

```
PROC PRINT:

     ID  PROVINCE CANTON DISTRICT IDNUM;
     VAR PROVINCE CANTON DISTRICT IDNUM FIELDNUM
         CROP ALTITUDE SOILTYPE
         RAIN_JAN RAIN_FEB  RAIN_MAR;
     TITLE INPUT DATA IN ORIGINAL ORDER;
```

(the remaining substitutions are left as exercises).

When the SAS compiler encounters the name of a MACRO (VARLIST or IDLIST in the example), it replaces the MACRO name by the macro text, i. e., by all the characters between the MACRO name and the "%" symbol. (See SAS79, page 12, for a brief description of MACROs.)

In the present example, the first PROC PRINT step of Figure 2.6 would be modified by the compiler to appear as in the first section of Figure 2.7. The words VARLIST and IDLIST have been replaced by the text of the MACROs. Note how the text of IDLIST is substituted in VARLIST as a part of the 'expansion'.

SAS does not automatically print the program text after substitution. One can specify that MACRO expansions are to be printed by placing an "OPTIONS MACROGEN;" statement in any place prior to the invocation of the option. (See SAS79, p.446).

Excercise:

Complete Figure 2.7. That is, write out the complete expansion of the text from Figure 2.6 as it would be produced by SAS.

7.2 Subroutines

SAS MACRO statements permit the programmer to insert the same text multiple times in his program. If the MACRO contains several statements (without statement labels) the original program text will be modified by the compiler so that, in effect, the same statements will be inserted at different places and possibly executed several times.

A subroutine, or subprogram, is a set of statements which appear only once in a SAS step but which may be executed several times from different places in the step. The SQRT function is an example of a built-in SAS subroutine. When the compiler encounters a SQRT in the program, it produces code to transfer control to the SQRT subroutine call.

In SAS79, one uses LINK and RETURN statements to program subroutines. (See SAS79, page 114, for a brief description of these statements.) As a simple illustration, suppose that certain complicated computations are to be performed at several places in a program, involving a variable X, a mean M, and a standard deviation S. The code shown in Figure 2.8 could be used for this purpose.

The first GO TO is necessary so that the subroutine will not be executed at the beginning of processing each observation.

The "name" of the subroutine is COMP, which is just the label of the first statement in the subroutine. Note the subroutine is called by a "LINK COMP;" statement, below.

Before a "LINK COMP;" statement, appropriate values are placed in X, M, and S, the subroutine's input arguments. After LINKing, the result is

Figure 2.8. An example of a Subroutine.

```
//SYSIN   DD *
  DATA A.B.;
        GO TO NEXT; * SKIP AROUND SUBROUTINE;
* SUBROUTINE TO COMPUTE COMPLICATED FUNCTION OF VARIABLE X,
MEAN M, AND STD.DEV.SE.  THE RESULT IS RETURNED IN RESULT;
COMP:
        Statement to perform computations
RESULT = ;
RETURN; * END OF SUBROUTINE COMP;
NEXT:
        LENGTH
        INPUT etc.
        X = AGE; M = MEANAGE; S = SD-AGE
        LINK COMP;
AGE-F = RESULT;
    X = WEIGHT; M-MEAN-WT; S-SD-AGE:
        LINK COMP;
        WEIGHT - F = RESULT;
```

moved from RESULT an appropriate variable. (AGE_ F in the first example).

Note that the statements in COMP have access to all current variables. Unless dropped, any variables used in COMP will be output. SAS subroutines are "internal subroutines", unlike Fortran subroutines, for example.

A subroutine in one SAS step is not available to other SAS steps. If it will be needed in several SAS steps, it should be made into a MACRO and inserted into each step by MACRO substitution. The subroutine is inside the MACRO:

```
MACRO  COMP-MAC
       COMP:
       RETURN; %
```

In each step where it is included, one would still use "LINK COMP;" to involve the subroutine.

7.3 Other RETURNs

Note there is another important use of the RETURN statement, not associated with subroutines. See SAS79, page 114, for a description.

II. 8. The INPUT Statement and Related Topics

(This section has been temporarily omitted).

9. Elementary SAS File Processing

The basic purpose of the Data Input Phase is to produce a SAS dataset which can then be manipulated with relative ease. Actually, one does not usually modify a SAS dataset which has already been created. "Manipulation" in a DATA step usually involves a repetition of the following steps:

a. Read an observation from an input SAS dataset.
b. Modify the observation. (Modify, add and/or drop variables; possibly delete the observation).
c. Write the modified observation to the output SAS dataset.

The process is repeated until all the observations from the input dataset have been processed.

Virtually all SAS data processing is some variation on this method. The following paragraphs discuss programming the basic method and some variations.

9.1 Basic Processing: DATA and SET

The most basic form of SAS dataset processing is described in the paragraphs above. A SAS step for this processing has the form:

```
DATA outputdsn;
 SET inputdsn;
 statements to compute new variables,
 drop variables, and/or
 delete observations
```

Where: outputdsn is the dataset name of the output SAS dataset inputdsn is the dataset name of the input SAS dataset.

One can optionally add the parameter END = varname to the SET statement, where varname is the name of a variable which the SAS supervisor will set to 0 for each input observation except the last and to 1 when the last observation is input.

(The SET statement is described in detail on pages 78-79 of SAS79.)

The statements following the SET statement are executed for each input observation.

Concatenating Datasets. Two or more input datasets may be concatenated by listing them, in order, in the SET statement. The general form is:

```
DATA  outputdsn;
 SET  inputdsn1  [(IN = var1)]
      inputdsn2  [(IN = var2)]
      inputdsnk  [(IN = vark)]
```

[(The square brackets indicate the use of the enclosed parameter is optional)].

As above, outputdsn is the name of the output dataset. The input dataset names are inputdsn1, inputdsn2, ..., inputdsnk. If specified, var1 is a Boolean variable which the SAS supervisor sets to 1 for each observation input from the first dataset and to 0 for each observation from the other datasets. Similarly var2, if specified, is set to 1 for each observation from the second dataset and to 0 for observations from other datasets. The other "IN" variables are similar.

As an example, if the dataset B.MALE contains data on male subjects and B.FEMALE contains data on female subjects, one could create a dataset containing data on both as follows:

```
DATA B.BOTH;
        SET
           B.FEMALE (IN=IN_FEM)
           B.MALE   (IN=IN_MALE);
        IF IN_FEM THEN SEX='F'
                  ELSE SEX='M';
```

The data for females would preceed the data for males.

9.2 Deleting Observations: Subsetting IF; DELETE

The DELETE statement is used to delete observations from the output dataset. The actual effects of executing a DELETE are to cause SAS:

a. To cease processing the current observation;
b. Not to write the current observation to the output dataset;
c. To immediately begin processing the next observation.

For example, if the dataset B.BOTH has data on males and females, one wishes to create a dataset containing data on males only, and B.BOTH contains a variable SEX, coded as 'M' or 'F' the following SAS step would be appropriate:

```
DATA B. FEMALE;
        SET B.BOTH;
        IF SEX NE FEMALE THEN DELETE;
```

The subsetting IF statement can be used to accomplish the same effect. The general format is: "IF expression;" where expression is a Boolean expression. The action of the statement is exactly equivalent to:

```
IF  expression THEN;
      ELSE DELETE;
```

(Note that the statement following THEN is a null statement.) The example above of selecting female observations could be coded:

```
DATA B.FEMALE;
      SET B.BOTH;
      IF SEX = 'F';
```

The subsetting IF is described on page 104 of SAS79. DELETE and Subsetting IF work in any DATA step.

9.3 Selecting a Subset of Variables: DROP, KEEP

In a DATA step one may choose to have SAS not write selected variables to the output file. SAS sets aside memory location for all variables defined for the step. For DATA...INPUT step this includes all variables in the INPUT statements plus all additional variables defined in statements within the step. For other DATA steps (DATA...SET, etc.) this include all variables in the input datasets plus any additional variables defined in the step. Because the memory locations are assigned to all defined variables, all defined variables are available for computation at all points in the step after they receive their values. (Variables input from a SAS dataset receive their values 'at' the SET (or MERGE or UPDATE) statement. Variables input via an INPUT receive their values upon execution of the INPUT statement. A variable appearing on the left of an assignment statement receives its value when the assignment statement is executed).

DROP and KEEP statements, described on page 111 of SAS79 are used to restrict output to a subset of the variables. Use form 'DROP droplist;' to specify that the variables in droplist are not to be written to the output dataset. Use "KEEP keeplist;" to specify that only the variables listed in

keeplist are to be written to the output dataset. (Examples of DROP and KEEP are shown in SAS79). Do not use both DROP and KEEP in the same step. DROP and KEEP do not affect storage in memory; they affect only the list of variables to be written out.

9.4 SORTING: PROC SORT

Sorting is especially easy in SAS, once the data are in a SAS dataset. Actually, rather than sorting a dataset, one specifies both an input SAS dataset and an output SAS dataset. SAS uses external software to read the data from the input dataset, sort the data using temporary datasets, and then write the sorted data to the output dataset. The general format is:

```
PROC SORT DATA = inputdsn
          OUT = outputdsn;
       BY key variables;
```

(The PROC SORT statement has additional options not discussed here; see SAS79 pages 373 et. seq.) Here, inputdsn and outputdsn denote the names of the input and output SAS datasets, respectively. If outputdsn is the same as inputdsn, the output is placed into a new SAS dataset. After the sort step successfully completes the sorting and writing, the output dataset is given the name of the input dataset and the input dataset is deleted.

If the sorting process fails, due to lack of disk space or lack of time, for example, the output dataset would not be completed, the name would not be transferred, and the effect would be the same as if the sort had never been attempted.

Except for small, temporary datasets it is poor programming practice to give the input and output datasets the same name.

Sorting Order. The PROC SORT step must always include a BY statement, which lists 'key' variables that determine the ordering of the output dataset.

At this point the reader sould carefully examine the SAS79 PROC SORT example, page 375. The key variables in that example are CITY and CHAPTER. CITY is the major key variable CHAPTER is the minor key variable. Note that observations are ordered by CHAPTER within CITY.

Unless otherwise specified, SAS sorts data so all variables in the sort BY list will be in ascending order. The values of the first BY variable will be in strictly ascending order. The values of the second BY variable will be grouped, each group being such that all observations in the group have the same value of the first BY variable. Within a group, the values of the second BY variable will be in strictly ascending order. (The example on page 375 of SAS79 illustrates this for the case of two BY variables.)

If more than two BY variables are used, the values of the (K + 1)-st BY variable are in order within each group defined by holding constant the values of the first K BY variables.

Data may be sorted into descending order on one or more BY variables by placing the keywork DESCENDING after such variables in the BY statement as, for example:

```
BY HEIGHT DESCENDING
   WEIGHT DESCENDING
   AGE;
```

Here, the sorted dataset values of HEIGHT would be in descending order. For a group of observations having the same value of HEIGHT, values of WEIGHT would be in descending order. For a group of observations having the same HEIGHT and WEIGHT, values of AGE would be in ascending order. The reader is urged to create his own examples by selecting a sample dataset with several discrete variables(e.g. race, sex, etc.), sort the data several times using different BY lists and different combinations of ascending and descending. Use PROC PRINT to list the dataset after each sort to observe the results.

9.5 "FIRST" and "LAST" Variables

In a DATA step with a SAS input data set (e.g., DATA...SET), if the input dataset is sorted it is convenient for the programmer to be able to determine when the value of one of the sort BY variables changes, i.e., to

determine:

a. If one of the BY variables changed value when the current observation was input (so that the current observation is the first with this value of the BY variable); or

b. If one of the BY variables will change when the next observation is read, i.e., if this is the last observation with this value of the BY variable.

SAS provides a mechanism for making these determinations. To activate the mechanism one uses a BY statement following the SET (or MERGE or UPDATE) statement, as follows:

```
DATA outputdsn;
     SET inputdsn;
     By key variables;
```

The key variables are listed just as they appeared on the PROC SORT: BY list, including the DESCENDING parameter, if used.

As an example, consider

```
PROC SORT DATA = B. CAFEORIG
          OUT = B.CAFESORT;
     BY PROVINCE CANTON DISTRICT;
DATA TEMP;
     SET B.CAFESORT;
     BY PROVINCE CANTON DISTRICT;
```

In this example SAS will generate 6 "automatic" Boolean variables, named:

FIRST.PROVINCE	LAST.PROVINCE
FIRST.CANTON	LAST.CANTON
FIRST.DISTRICT	LAST.DISTRICT

In general two automatic, Boolean variables are generated for each variable in the BY list. The values of these variables are set as follows:

FIRST.var = { 1 if the value of var, or any variable which preceeds var in the BY list, changed when the current observation was input.
0 otherwise

LAST.var = { 1 if the value of var, or any variable which preceeds var in the By list, will change when the next observation is input
0 otherwise

As an example, consider the following table. The values of the key variables are shown on the left, and the values of automatic variables are shown on the right. This "dataset" was generated by a program segment such as the one in the box above:

OBS	REGION	CANTON	DISTRICT	First REGION	Last REGION	First CANTON	Last CANTON	First DISTRICT	LAST DISTRICT
1	1	1	1			1	0	1	0
2	1	1	1			0	0	0	1
3	1	1	2			0	0	1	1
4	1	1	3			0	1	1	1
5	1	2	1			1	1	1	1
6	1	3	1			1	0	1	0
7	1	3	1			0	1	0	1
8	1	4	1			1	0	1	1
9	1	4	2			0	0	1	0
10	1	4	2			0	1	0	1
11	2	4	2			1	1	1	1

Note that in observation 5, LAST DISTRICT = 1 even though DISTRICT = 1 in both observation 5 and observation 6. Since DISTRICT is sorted within CANTON, whenever CANTON changes DISTRICT is also assumed to change. Thus, observation 5 is the last observation for DISTRICT 1 within CANTON 2.

Similarly, when REGION changes, as from observation 10 to observation 11, both CANTON and DISTRICT "change", even though the values do not appear to change. This is because observation 10 contains data on CANTON 4 within REGION 1, which is assumed to be different from CANTON 4 and REGION 2 (observation 11). Since the CANTON is different, the REGION is also different.

Programming note: It is often desirable to know if there is only one observation for each combination of values of the key variables, i.e., if the values of the key values uniquely identify an observation. In the table above, for example, the key variables do not uniquely identify the observations; the first two observations have the same key values.

An observation is uniquely identified by the BY variables if and only if, for the last variable in the BY list, FIRST.last = LAST.last = 1. In the example above, the last variable in the BY list is DISTRICT; only those observations with FIRST.DISTRICT = LAST.DISTRICT = 1 are uniquely identified by the BY variables. (These are observations 3, 4, 5, 8 and 11).

The UPDATE statement, discussed below, requires that the "master" file have all observations uniquely identified by the values of the BY variables. The technique above can be used to check for this condition.

9.6 A Summarization Example:

The ideas and techniques discussed in the preceeding paragraphs are illustrated by the program segment in Figure 2.10. In this example the input dataset is B.CAFESORT, which is sorted BY PROVINCE CANTON DISTRICT IDNUM. It is desired to compute the total of the variable QUANTITY, and the average of the variables WATER, IN_DIST, OUT_DIST for each DISTRICT (within CANTON and PROVINCE).

Figure 2.10 An illustration of the Use of "FIRST", and "LAST". Variable for a Sorted SAS Dataset.

```
DATA B.STATS;
    SET B.CAFESORT; * NOTE:  CAFESORT IS SORTED--;
    BY PROVINCE CANTON DISTRICT;
    * "RETAIN" THE VARIABLES USED FOR SUMMING AND COUNTING;
    RETAIN N TOT_QTY SUM_H20 SUM-IND SUM_OUT;
    * ON THE FIRST OBSERVATION IN A DISTRICT
    INITIALIZE THE TOTALS AND N;
    IF FIRST  DISTRICT
                    THEN DO;
                    N = 1
                    TOT_QTY=QUANTITY;
                    SUM_H20=WATER;
                    SUM_IND=IN-DIST;
                    SUM_OUD= OUT_DIST;
    end;
    ELSE DO;
  * FOR SUBSEQUENT OBSERVATIONS INCREMENT TOTALS;
    N=N+1
    TOT_QTY = TOT_QTY +QUANTITY;
    SUM_H20 = SUM_H20 + WATER;
    SUM_IND = SUM_IND + IN-DIST;
    SUM_OUD = SUM_OUD + OUT-DIST;
    END;
```

Fig. 2.10 Contin...

```
* AT THE LAST OBSERVATION FOR THE DISTRICT,
  COMPUTE MEANS AND OUTPUT AND OBSERVATION;
  IF LAST, DISTRICT
     THEN DO;
          MEAN_H20 = SUM_H20/N;
          MEAN_IND = SUM_IND/N;
          MEAN_OUD = MEAN_OUD/N;
          OUTPUT;
END;
ELSE DELETE; * WE OUTPUT AN OBSERVATION
               ONLY AT THE END OF THE
               DISTRICT;
KEEP PROVINCE CANTON DISTRICT N
  TOT_QTY MEAN_H20  MEAN_IND MEAN_OUD;
```

The output dataset, B.STATS, is to contain the variables PROVINCE, CANTON, DISTRICT, TOT_QTY (total of QUANTITY), MEAN_H20, MEAN_OUD (means of WATER, IN_DIST, AND OUT_DIST, respectively), and N, the number of observations used to compute the sum and means. It is assumed there are no missing values.

Exercise. Invent data values for the variables WATER, QUANTITY, IN_DIST for each of the observations shown in the last box in section 9.5. By hand (not on the computer!) trace through the execution of the program in Figure 2.10 and 'produce' the output dataset. Verify that you understand exactly how the various features of the program work.

Note: There are more efficient ways to perform the task of Figure 2.10 in SAS, including PROC SUMMARY and PROC MEANS. The purpose of the example is to illustrate the features discussed in this chapter and related programming techniques.

10 Advanced SAS File Processing: MERGE, UPDATE

The reader is now sufficiently prepared to study chapter 9 of SAS79, which discusses the SET, MERGE and UPDATE statements in detail. The SAS79 chapter 9 description is adequate for the purposes of this section.

11. File Manipulation PROCs

SAS provides several PROCs for assisting in the manipulation of OS files and SAS datasets. The student should now study the following section of SAS79:

Chapter 14, 'Overview: The PROC Step". p. 119
PROC CONTENTS, p. 179
PROC COPY, p. 171
PROC DATASETS, p. 179
PROC DELETE, p. 181
PROC MEANS, p. 303
PROC PRINT, p. 353
PROC RELEASE, p. 365
PROC SUMMARY, p. 397
PROC TAPECOPY, p. 419
PROC UNIVARIATE,p.427

The statistical procedures in the list above, MEANS, SUMMARY and UNIVARIATE are useful for reviewing a dataset to scan for bad data values, etc. In addition, MEANS and SUMMARY are useful for summarizing a dataset i.e., inputting one dataset and computing summary statistics which are then output to another dataset. The output dataset is then available for further processing.

III. A GENERAL RESEARCH DATA MANAGEMENT SYSTEM MODEL

1. Introduction: Objectives

The purpose of this section is to present a general model of an RDM system. The word 'model' does not mean that the system is ideal and could be reproduced exactly for each problem which arises. Rather, the system is a model in the sense that it has most of the features one would consider for inclusion in a system being designed. For a particular problem, one must select appropppriate parts of the system and adapt those parts to the needs of the particular problem.

The following sections contain a flowchart of the model system and description of all the subsystems and corponents. Implementation of the system in SAS is discussed in those sections where the methods of implementation may not be obvious.

2. Flowchart and Major Components of the Model System

A flowchart of the model system is presented in Figures 3.1 and 3.2.

The system has five major components:

- Preliminary Manual Processing
- The Main Data Management System (Error Detection, File Maintenance)
- The Inventory Subsystem
- The Error Correction Subsystem
- The Quality Control Subsystem (Not in Flowchart)

In the flowchart, the Main Data Management System (DMS) is indicated by a box made of broken(---) lines, Figure 3.1. Preliminary Manual Processing consists of those steps above the DMS box in Figure 3.1. The Inventory Subsystem is to the left of the Main DMS box, and the Error Correction Subsystem is diagramed in Figure 3.2. The following sections contain detailed discussions of these components.

3. Distributed Flat File Processing: Helms (1979) discusses the 'Distributed Flat File' strategy for RDM problems with multiple data streams. The basic points are as follows.

FIGURE 3.1 FLOWCHART OF A TYPICAL RESEARCH DATA MANAGEMENT SYSTEM

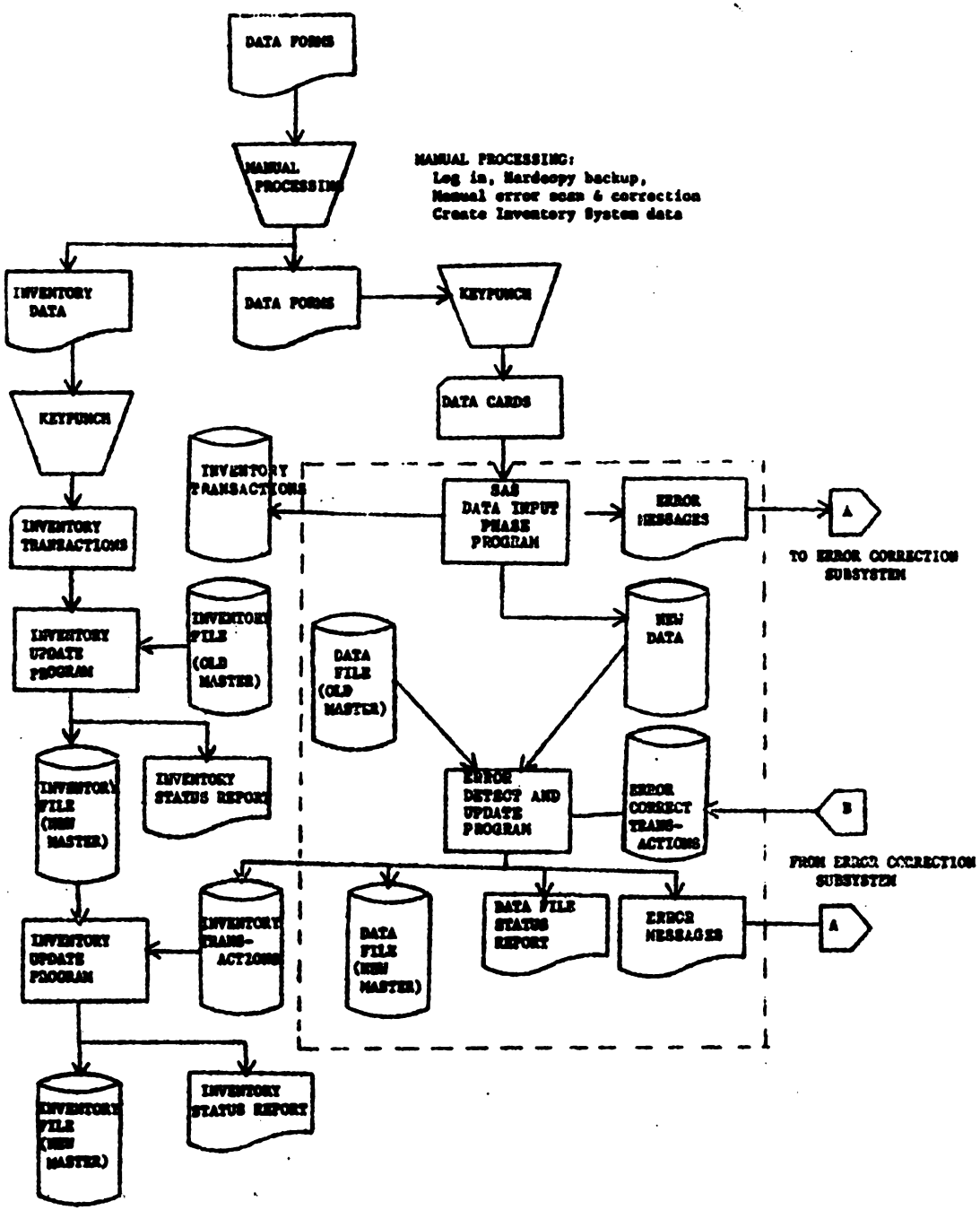
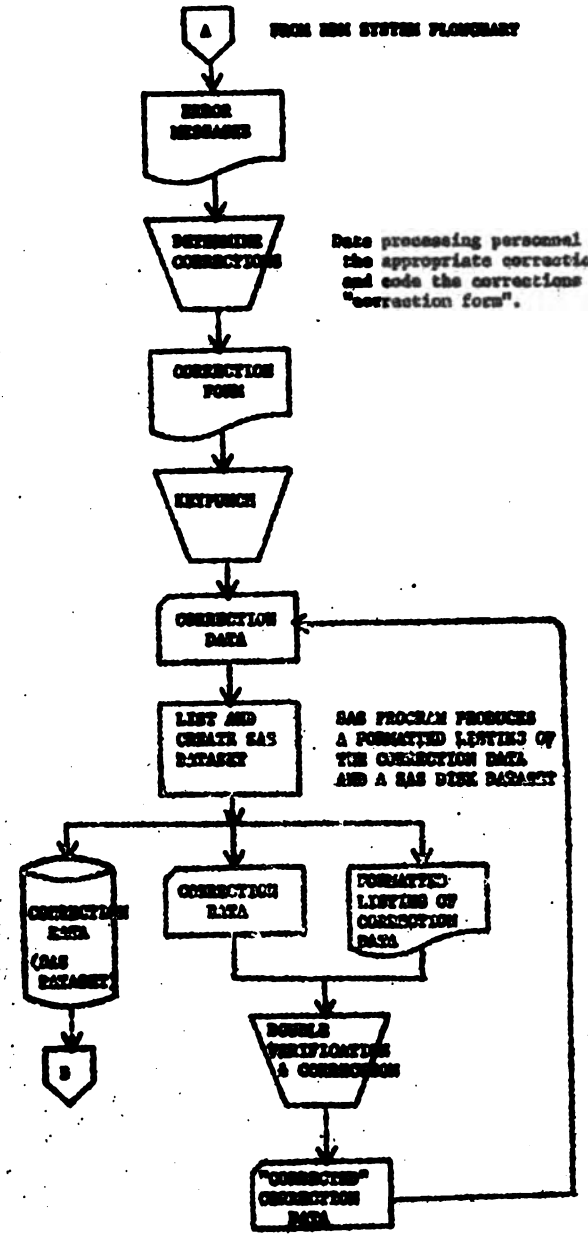


FIGURE 3.2 THE ERROR CORRECTION SUBSYSTEM OF THE TYPICAL RIM SYSTEM



Many projects involve the collection of several different types of data over a period of time. If the different data types (data forms) are all collected on the same schedule then the data are easily linked by time period and all of the data with the same schedule constitute one data stream. However, in many cases different types of data are collected on different schedules. For example, one study may have one type of data collected monthly, another type collected quarterly (each three months), another type collected annually, and yet another type collected when a particular event occurs, i.e., essentially at random. In such a case, each type of data constitutes one data stream; the whole project has several data streams. For the production data processing phase, Helms (1979) has shown that setting up a separate RDM system for each stream is often an excellent strategy. This is called the 'Distributed Flat Files' (DFF) strategy.

If one takes the DFF strategy, then instead of designing and implementing one large and complicated system, one implements one much simpler system for each data stream.

The model system described in the following sections does not assume either the DFF strategy or the converse, the 'monolithic system' strategy. To use the DFF strategy one simply implements separate systems for separate data streams. The monolithic system strategy may be implemented by designing a DMS component capable of handling all the streams simultaneously.

4. Preliminary Manual Processing:

The tasks involved in preliminary manual processing depend to a great extent on local factors such as the source of the data (internal or external to the organization), whether data forms are completed locally or at another location. The nature of the study (longitudinal or cross-sectional), etc.

4.1 Log in

Forms are typically received in a package, which may contain several types of forms from several data streams. One of the first manual steps is to 'log in' the forms, making a paper record of the existence and arrival

of each form. Typically, this 'log in' step creates data for the Inventory Subsystem, which maintains information about the presence, location and processing status of data records. Inventory data are keyed immediately. Project data undergo additional processing before keying.

4.2 Hard Copy Backup:

If the data have any substantial value the next step is to make a 'hard copy backup' copy of the data forms. For very small projects the backup may be by xerography or other photocopying. For larger studies microfilm is less expensive and requires less storage space, though there is a greater initial investment for the equipment. Another advantage of microfilm is that additional backup copies are very inexpensive; and it is practical to obtain and to store an additional copy in a remote, secure location. In the case of a fire at the data processing facility, the back-up at the remote location may be the only remaining copy of the data.

MANUAL ERROR SCAN AND CORRECTION

Before keying, the data forms are reviewed carefully by a person familiar with the project and its data. Correcting error at this stage is much less expensive and less troublesome than making corrections to a computer file! The reviewer carefully checks each form for legibility, for complicance with instructions, for obviously incorrect or inconsistent data, for missing data, etc.

The most important checking in the manual error scan is for key field errors. A key field is a data field which will later be used for sorting and/or for identifying a particular record, a particular subject, etc. Key fields are often called identification fields and often appear in BY statements in SAS program. Non-key fields are called 'data fields'.

Key field errors are much more difficult to correct than data field errors once the data are on a computer file. The procedure for correcting a data field error is to input a new, correct value and update the observation (described later). To 'correct' a key field it is necessary to make

an entire new observation (keypunch new cards) add the new observation to the file, and delete the old observation with the erroneous key values.

The manual data reviewer should very carefully key field values and determine that they are absolutely correct. It is sufficient to simply determine that key field values are in a valid range.

Any errors detected in the manual error scan are corrected before data are keypunched. Usually, the data reviewer must contact the persons who entered the data on the forms in order to determine corrections.

New errors may be accidentally introduced when "corrections" are made. All data changes made by the reviewer are checked by a supervisor before the corrections etc., are entered into the system.

After the manual error scan and correction (if necessary) the data are sent for keying. Note that the inventory data, created by the log-in process were sent to be keyed immediately after log-in.

Key-and-Verity

Data are keyed (keypunch, key-to-disks or equivalent) and verified. Persons who process unverified data are wasting time and money---random numbers, the product produced by unverified keying, can be created much more quickly and less expensively by computer. If one wishes to process such random numbers, why bother with data forms and keying?

The output from the key--and--verify step is shown on the flowchart (Figure 3.1) as a data card. This is only symbolic; if one uses key-to-disk or key-to-tape an appropppriate symbol can be subtituted.

5. Emphasis on Security and Error Detection and Correction

The reader will quickly observe that one of the difference between amateur and professional data management is the professional's concern for the quality and security of the data. These concerns are manifest in extensive data backup and error detection--correction facilities in the system. In many research projects there is a point, after the data have been collected and before substantial analyses have been completed at

which almost the entire investment in the study -the financial investment, the investment of significant time from the careers of participating scientists, and the manpower investment-is concentrated in the data files on a few tapes or disks. Destroy those files and you destroy almost the entire investment in the study.

Data files may be destroyed due to physical disaster, such as fire or flooding or by computer malfunction (disk "crash", data overwritten by other data, etc.) Backing up files is the appropriate protection against this type of disaster, this topic will be discussed in the section on Data Security.

One can also effectively destroy the project's investment by introducing random (or systematic) errors into the data and not removing them. Errors are introduced at each stage of human transcription or processing. (Computers introduce errors, too, but that is a different topic.) Errors are introduced when the data are originally 'captured' (as responder error in interviews), when forms are completed, when "corrections" are made. The question is not <u>whether</u> errors are introduced at each of these stages, but <u>how frequently</u>. That is, the question is: what is the error rate of each human transcription or process?

If the cumulative rate for uncorrected errors is too high one destroys the project's investments in the study just as surely as if one had burned all the copies of the data. Destruction by errors is even more insidious than by fire. One is immediately aware of destruction by fire. When a file is destroyed by errors the results of the errors may be published in the scientific literature and the erroneous conclusions may not be discovered for years. The drug thalidomide, which caused so many birth defects, was marketed without sufficient experimentation for side effects, but it is obvious that uncorrected data errors could lead to this kind of disaster.

Data errors are important. The reader will note that a substantial component of the system is devoted to the detection and correction of

errors and the estimation of error rates (quality control subsystem).

6. Error Detection

The primary objectives of the main Data Management System (DMS) are:

1. To detect errors and provide for their correction
2. To maintain (create, update) the master data file
3. To produce reports to project managers on the status of data processing.
4. To provide for data security through systematic file backup.

A system may be implemented in one, or several computer programs. In a SAS implementation, such a 'program' usually includes several SAS steps.

Error detection is performed in two places in the Flowchart of Figure 3.1: in the Data Input Phase program and in the Error Detect and Update program.

The principal purpose of error detection in the Data Input Phase program is to detect structural errors, such as invalid key field values and 'missing cards'. The system's major computerized error detection effort is concentrated in the error detection component of the Error Detect and Update program. This component subjects data to the most rigorous tests practical, within the framework of the particular project.

The remainder of this section describes techniques of error detection. There are two basic types of error detection tests, field tests and consistency tests.

A field test is a test based solely on knowledge about values which are permissable for the data field (variable). Three types of field tests are commonly used: 1. valid values, 2. valid range, and 3. field type definition.

6.1 Valid Values Field Tests. One uses a valid values field test when the complete set of valid values for a variable (field) is known and has few elements. Consider, for example the item from a data form:

```
17.  Sex of subject:
     1.  Male ...M
     2.  Female..F
```

There are only two valid values for the variable, SEX, from this item. The valid value field test would determine if the data value is one of the valid value, e.g.,

```
IF NOT (SEX = 'M' or SEX = 'F')
  THEN  take error action;
```

We have not yet discussed 'take error action', but at least one example has been illustrated, Another form of this statement uses a Boolean variable:

```
ERR1 = NOT (SEX = 'M'
         OR SEX = 'F')
```

This type of statement can be 'mechanized' (for easy program keypunching) for variables with more than valid values. Suppose ITEM has valid values 1, 2, 3, 4 and 9; the following statement contains the essential part of valid value test:

```
ERR2 = NOT (ITEM = 1
         OR ITEM = 2
         OR ITEM = 3
         OR ITEM = 4
         OR ITEM = 9);
```

Valid value testing tends to produce long SAS programs for obvious reasons. Proper program formatting improves readability, reduces errors, and cuts debugging/testing time.

6.2 Valid Range Field Tests. Numeric variables which are known to lie in a particular range, and which have more valid values than can can reasonably be checked by a valid values test may be checked by a valid range test.

As an example, consider a date which is known to lie between 20OCT73 and 05JAN75 and for which the year, month and day have been separately input into the numeric variables YEAR, MONTH and DAY respectively.

If a date is subject to error, it is good programming practice to input the components separately and to perform the conversion to a date variable internally, where the program can control the effects of errors. Errors in SAS date format conversion result in a missing value for the date -- a situation beyond the programmar's control. The program segment in Figure 3.3 performs valid range tests on MONTH, DAY and YEAR and, if the individual values are valid, converts these values to a date variable and checks the range of that. Note that the program leaves detailed checking of the number of days in the month to the SAS function MDY. (See SAS79, page 41) MDY will return a missing value if the combination of month and day are invalid.

6.3 Consistency Tests:

A field test compares the value of a variable with the allowable values of that variable, other factors being ignored. A consistency test uses comparisons of the values of two or more variables. Dates provide a good example. Given a MONTH, DAY and YEAR, as in Figure 3,3, a field test for Day must use only the valid range, 1 DAY 31. A consistency test of MONTH and DAY can further check for invalid combinations, such as February 30. Thus, DAY = 30 would pass the valid value field test, but the combination MONTH = 2 and DAY = 30 would fail

the consistency test. (The classic example in medical data is a comparison of SEX versus PREGNANT, to check for pregnant males).

The information for a consistency test may be know a priori from certain rules, as in the date example, or from previously published results. In addition, consistency tests can be generated from data. Consider two numeric variables, X and Y which have a high linear correlation coefficient. One can perform a regression analysis, to obtain regression coefficients and 99.9% confidence bands for a future observation of Y, given a value of X.

The test consist r in determining for a given X value, whether or not Y lies in the 99.9% confidence interval. If Y is inside the confidence interval the test is passed, otherwise the test is failed. Of course, one can use confidence coefficients other than 99.9%.

The data for determining the consistency tests may come from preliminary analysis of the data being processed, from prior data, or from previously published results for similar data.

6.4 Errors and Improbable Values. The process of error detection is inexact. For example, a date which should be (in yymmdd format) 800105 and is actually entered as 800205 might not be detected by any test. In contrast, a rare value such as a HEIGHT = 225 cm for a human female, ought to fail a valid value or consistency test, even if the value is correct.

The objectives of error detection strategies and procedures are:

1. To detect as many errors as possible
2. Consistent with a reasonable processing cost.

In a valid range test for human HEIGHT, for example, by making the valid range very short, virtually all the very large errors (difference between true height and recorded height) will be reported, but a large number of correct values will be reported as possible errors. It costs money to check an error message and update the file to reflect the results

of the check.

Reasonable error detection necessarily involves trade-offs between the cost of not detecting errors which exist in the data and the cost of processing error messages for correct data.

The appropriate approach is to define tests which detect a reasonable proportion of improbable values, whether correct or not. Professional judgement is required to determine the proportion of correct values which will be reported as 'errors' (improbable values).

Redundant Information: All error checking is based on partially or totally redundant information. In the example of valid value testing of the variable SEX, the two redundant pieces of information are:

1. The value of SEX
2. The valid values of SEX

For another example, since human height and weight are highly correlated, these variables are partially redundant and can be used for a consistency test. Variables which are totally unrelated cannot be used for a meaningful consistency test. The more redundant the information, i.e., the closer the relationship between two (or more) variables, the better the resulting test will be.

The principle of basing tests on redundant information is important in the design of data collection forms and instruments. Completely redundant data will be collected on variables which are crucially important to a study. Chemists, for example, routinely run three (or more) trials of an important assay. If one of the values differs markedly from the others, corrective action is taken. When such multiple determinations are made they should be included in the data.

We use only partial redundancy for less important variables. The redundancy may come from knowledge about the variable (as in a valid value test) or from knowledge of the approximate relationship between two or more values, as in a consistency test.

When designing data forms and a data management system, one must determine the relative importance of each variable to be collected. Totally redundant information must be collected on crucially important variables. Partially redundant information is collected on variables of less importance. Variables which are not important enough to justify some redundance and careful checking should be omitted from the study.

7. What to do with "Dirty" Data

What should the system do with observations which contain detected errors or "improbable values"? There are two basic strategies.

Clean File -- Dirty File. One strategy for dealing with the problem is to produce two "master" files. All observations which have no detected errors are placed in a "clean" file. All observations containing detected errors, or improbable values, are placed in a "dirty" file. Error corrections are made to observations in the dirty file. When an observation has been "purified" (all detected errors corrected and all correct-but-improbable values verified), it is moved to the clean file in an update run.

The clean file -- dirty file strategy has the advantage that preliminary analyses can be performed with the clean file with no particular precautions about erroneous data.

In some projects, however, error correction can require substantial time especially if the data volume is large and the data collection center is remote from the data processing center. In such a case, data accumulate in the dirty file and movement to the clean file is very slow. In addition, if an observation contains many variables, the probability is great that at least one variable will fail at least one test, whether the value is improbable - but - correct, or in error. Here again, observations tend to accummulate in the dirty file and move slowly to the clean file. The failure of relatively unimportant variables to pass tests may keep correct, important variables out of the clean file for months.

Status Bytes: The alternative strategy, apparently first used for research data by Helms (1972) utilizes a single master data file which contains all available data records. Each record ("observation" in SAS) contains one "status indicator" for each data variable. The indicator contains information about the quality of the associated variable. (For convenience, systems using this technique usually use one byte to store the value of each status indicator, and the term 'status byte' has become popular. More compact schemes are possible because the number of possibly statuses is small).

The Error Detect and Update program sets the values of status bytes. At the beginning of error testing of a new observation, all status bytes are initialized to values indicating, 'no error detected'. (See figure 3.4 for a typical set of status byte codes). It a variable fails a valid-value, valid-range or consistency test, the status byte is set accordingly (and error messages are printed).

Figure 3.4. A Typical Set of Status Byte Codes

Code	Interpretation
'Ø' (blank)	No error detected for this value
'ø'	This value failed one or more tests but was subsequently determined by a human to be correct. The system will not change this code.
'c'	This value has failed a consistency test.
'M'	The data value is missing.
'V'	This value failed a valid value or valid range test.

Note: In the IBM 370 collating sequence,

'Ø' < 'ø' < 'c' < 'M' < 'V'

This ordering of the codes reflects increasing severity of 'error' or decreasing estimated quality of the value.

Status bytes maybe reset as a result of the error correction and update process, described in a subsequent section.

With the status byte technique all the data are available for preliminary analysis. The analyst can determine, on a variable-by-variable basis, how 'clean' data should be to enter the analysis. If there are errors in variables not involved in the analysis, the 'clean' data in the observation may still be used. In contrast, under the clean file -- dirty file strategy, only records which are completely clean may be used.

Neither strategy is always superior; the strategy to choose will depend upon the circumstances of the particular project.

8. File Maintenance

Although essentially distinct, the file maintenance function and error detection function are performed in the same program, Error Detect and Update. For file maintenance one could choose either direct or sequential access to master files. SAS supports only sequential processing, which is discussed here.

The basic file maintenance process is based upon the SAS DATA and UPDATE statements. The procedure involves copying data from an "old master" file (the most up-to-date version of the data available for input) to a "new master" file. Modifications are made during the copying process as indicated by 'update transactions' (observations) from a 'transaction file'.

All three files contain the same key (BY) variables and all are sorted in the same sequence. The update procedure is described in some detail in chapter 9 of SAS79 especially pages 85-88.

9. The Error Detect and Update "EDUPDATE" Program

The key software in the system is the Error Detect and Update (EDUPDATE) Program. This program may be defined in terms of its inputs, its outputs, and its processing, which are discussed in the following

paragraphs.

9.1 Inputs

As shown in Figure 3.1. The inputs to the EDUPDATE are the NEW DATA ("NEWDATA"), ERROR CORRECT TRANSACTION ("ERRCORR") and OLD MASTER ("OLDMASTR") files.

Since a SAS DATA---UPDATE step can process only one old master and one transaction file, a preliminary SAS step is executed similar to:

```
DATA TEMP;
     UPDATE B.NEWDATA
            B.ERRCORR;
     BY key variables;
```

The temporary dataset TEMP is then used as input to main step of EDUPDATE. However, it is more convenient to describe NEWDATA and ERRCORR separately.

NEWDATA, This dataset contains new data, sorted by the key variables. The Data Input Phase ("DIP") program has already verified that:

a. Key variables pass appropriate field tests.
b. Each record on the dataset is uniquely identified by its key variables.

Program DIP also adds one variable (to key and data variables): INPUTDAY, containing the date and time the DIP parogram was executed.

ERRCORR. The format of this SAS dataset will depend to a great extent on the details of the error correction subsystem, described in a separate section. If a "turnaround" keyable error correction form, described in that section, is used, the ERRCORR dataset will typically be as follows.

Each observation contains all key, data and status byte variables with the same types and lengths as on the OLDMASTR. The observation also contains the following additional variables:

CORRTIME, A SAS date-time variable containing the date and time the error correction DIP program was run.

FLAG, a one-byte character variable, input from the error correction form, coded as follows:

'C' indicates this observation contains a correction data value to replace the value of a variable on OLDMASTR.

'D' indicates that the observation on OLDMASTR with key values matching this record is to be deleted.

'B' indicates that this is a new data observation. New data observations enter via the correction stream when, for example, an error is discovered in key values. The old record is deleted and a new observation, with correct key values, is entered.

Typically, one ERRCORR observation is generated for each data correction. For example, if three fields are being corrected for one OLDMASTR observation, ERRCORR would contain one observation for each. Thus, in one ERRCORR observation most of the variables would have missing values. The Error Correction Subsystem might combine all ERRCORR observations for one OLDMASTR observation into a single observation.

ERRCORR is sorted by the key variables and FLAG.

Note that CORRDIP, the Correction Data Input Phase program, stores both data values and status bytes on ERRCORR. The names of the ERRCORR status byte variables are different from OLDMASTR status byte variables so that the status bytes of OLDMASTR must be explicitly updated by EDUPDATE.

(Otherwise blank status bytes from ERRCOR would overwrite non-blank status bytes from OLDMASTR during the SAS update phase).

OLDMASTR is the most recently created version of the master data file. It contains all the key, data, status, and date-time variables described above plus:

UPDATED, a date-time variable set to the date-time that an observation is added to OLDMASTR or modified by EDUPDATE.

FLAG is not included in OLDMASTR or NEWMASTR..

CLEAN a Boolean variable with value 1 if the observation is 'clean', or otherwise. "Clean" means all variables have status <= '¢'

CHECKED, a date-time variable indicating the date-time error detection was performed.
Of course, OLDMASTR is sorted by the key variables.

9.2 Outputs

EDUPDATE creates two output SAS datasets, NEWMASTR and INV_TX, and two or more print datasets, including MESSAGES and ERRORS.

NEWMASTR contains the same variables as OLDMASTR and is sorted in the same sequence.

INV_TX, the Inventory File Transaction dataset, contains one observation for each:

a. New observation added to the master file;
b. Master file observation modified by EDUPDATE.

INV_TX contains the key variables, plus the variables UPDATED, CLEAN, and CHECKED. The use of this file is described in the Inventory Subsystem section.

The ERRORS print file contains data-error messages for 'errors' detected by EDUPDATE. This printout is described in the Error Correction

Subsystem section.

MESSAGES. This print file contains a report on the updating process, including:

Total number of observations in OLDMASTR
Total number of observations deleted
Total number of new observations added
Total number of observations in NEWMASTR

(These totals permit a determination of whether observations have been 'lost' or erroneously added between or during update runs.) MESSAGES also contains the total number of 'clean' observations, total number of corrections, and other interesting statistics.

DELETES. This print file contains a formatted listing of all observations deleted during the update run.

PROBLEMS. This print file contains formatted listings of observations involved in apparently erroneous transactions as, for example:

- When a correction or deletion observation appears in ERRCORR with no matching observation on OLDMASTR.
- When a new data observation updates an observation previously on OLDMASTR.
- Only correction or deletion observations should update an observation on OLDMASTR.

Other update problems are also reported on this file.

9.3 EDUPDATE Processing:

As noted in the section on EDUPDATE inputs, EDUPDATE contains a preprocessing step to combine NEWDATA and ERRCORR into one dataset TEMP.

The second SAS step is typically the update step, with a basic structure indicated by the following:

```
DATA B.MASTER14
        INVENTOR TX27 (KEEP = KEY UPDATED CLEAN CHECKED);
UPDATE B.MASTER13 (IN = IN-OLDM);
        TEMP      (IN = IN-T);
       * MASTER14  IS NEWMASTR;
       * MASTER13  IS OLDMASTR;
       * TX27      CONTAINS OUTPUT INVENTORY FILE TRANSACTIONS;
       * TEMP      CONTAINS NEWDATA AND ERRCORR;
```

This is a section of code checking for, and handling, processing problems. See PROBLEMS file description.

This block represents a section of code which processes a DELETE observation

This block represents a section of code which processes correction updates, including moving new status byte info. into MASTER status byte variables.

This block represents a section of code which performs field tests on new or modified fields.

This block represents a section of code which performs consistency tests on new or modified field. Note: program will not modify a 'C' status.

This block represents code which outputs observations to NEWMASTR and TX, as appropriate.

Note that the error detection blocks, upon detecting an 'error', print appropriate error messages and set status bytes to appropriate values. Fields which are unchanged from OLDMASTR are not field-tested. A consistency test is performed only if at least one of the participating variables has a new value and all have passed field tests.

Backup, File Rotation and Naming of MASTER Files:

The figure above illustrates the use of numbers in creating dataset names for MASTER and transaction files. Note that the name actually used for OLDMASTR is B.MASTER13 and the name used for NEWMASTR is B.MASTER14. This indicates that OLDMASTR was a result of the 13-th update and the current update is the 14-th, producing MASTER14. In this case both MASTERS are in the same database, B. (A better strategy uses two databases on separate disk packs, especially for large datasets.)

The OLDMASTR is retained as a backup. If NEWMASTR is destroyed for some reason, or if the update job fails the first action to be taken is to copy (OLDMASTR) (here B.MASTER13) to tape and place this tape in a safe place. This action protects OLDMASTR as a backup, in case further problems occur. Then, after the backup is successfully completed, attempts are made to run the update correctly.

If, in copying to tape, OLDMASTR should be destroyed (as a result of system error, power failure during the run, or whatever cause) an attempt should be made to make a backup copy of the previous OLDMASTR (here B.MASTER12). Then two update runs would be made, one to re-produce OLDMASTR and then one to produce NEWMASTR.

From time to time the NEWMASTR is copied to a tape which is placed in a safe, remote location. Many institutions do this after each eight updates (MASTER8, MASTER16, etc.)

Eventually, one exhausts the space in the SAS databases used for the MASTER datasets. (A PROC CONTENTS should be run before each update to determine if enough space is available. Space is made available by

detecting the oldest MASTER datasets and copying corresponding transaction datasets to tape. Before deleting any MASTER dataset, the most recent MASTER is copied to tape, along with all transaction datasets used to produce it. For example, suppose MASTER8 was previously saved to tape and updates have been run to produce MASTER9, MASTER10,..., MASTER14.

We now need to delete datasets to make space available for MASTER 15. First we copy to tape MASTER14 and all of the transaction datasets used to create MASTER9, MASTER10, ..., MASTER14. This tape is checked for readability by copying its contents to another tape. (PROC TAPECOPY). The extra copy is checked for readability and moved to a remote, safe location. Now, MASTER8, MASTER9, ..., MASTER12 (not MASTER13 or MASTER 14) and the transaction files used to create MASTER8, ..., MASTER13 can be deleted. We always keep on disk the two most recent MASTERs and the transaction files used to create them.

This 'insurance' system provides security for MASTER files against destruction. If a MASTER file in accidentally destroyed, it can be re-created with a reasonable amount of time, effort and cost.

11. The Inventory Subsystem

11.1 Introduction

The Inventory Subsystem (INVS) is a small data management system which helps data processing project management control its inventory of data records. The INVS maintains information about the progress of each data record through the various components of data management. If the main data management system loses records, or erroneously 'generates' additional records, this fact may be discovered by examination of INVS reports.

INVS receives information (transaction observations) from each of the following stages of data management:

- 'arrival' observations, from Preliminary Manual Processing

- 'Input' observations from the Data Input Phase Program
- 'Error' observations if error messages are generated for the data observation by either the Data Input Phase or EDUPDATE programs.
- 'Correction' observations generated by the EDUPDATE program when correction transactions are updated to the master file.
- 'Update' observations when the data observation is added to the data file.

Each INVS transaction contains the date (arrival observations) or datetime (computer-generate observations) of the event being reported. Thus, the progress of a data observation through the system can be traced through INVS data. (Incidentally, the same variables are maintained on the MASTER files).

A flowchart of this INVS is shown in Figure 3.6. There are three basic phases:

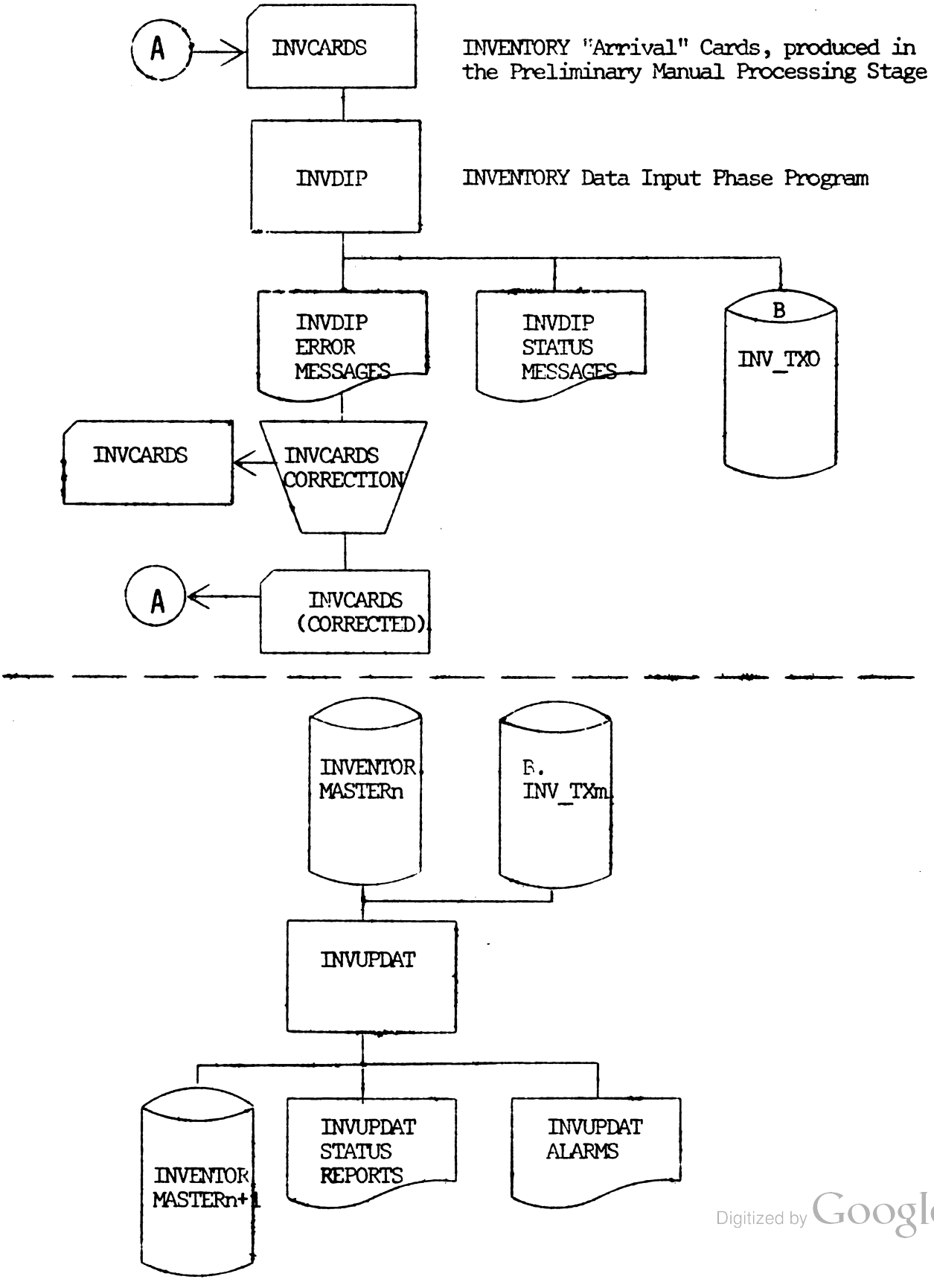
- Generating, checking and correcting arrival records in Preliminary Manual Processing.
- Generating other transaction records (by other subsystem computer programs, such as DIP, EDUPDATE).
- File maintenance and Report Generation.

The flowchart shows only the first and third phases; the second is a component of other subsystems.

11.2 Generating arrival observations.

The preliminary manual processing phase generates the information for arrival records (INVCARDS in the flowchart). The details of this component depend greatly upon the details of the particular project: the quantity of forms received and processed, the number of data streams, the personnel available, etc. The objective is to obtain very accurate in-

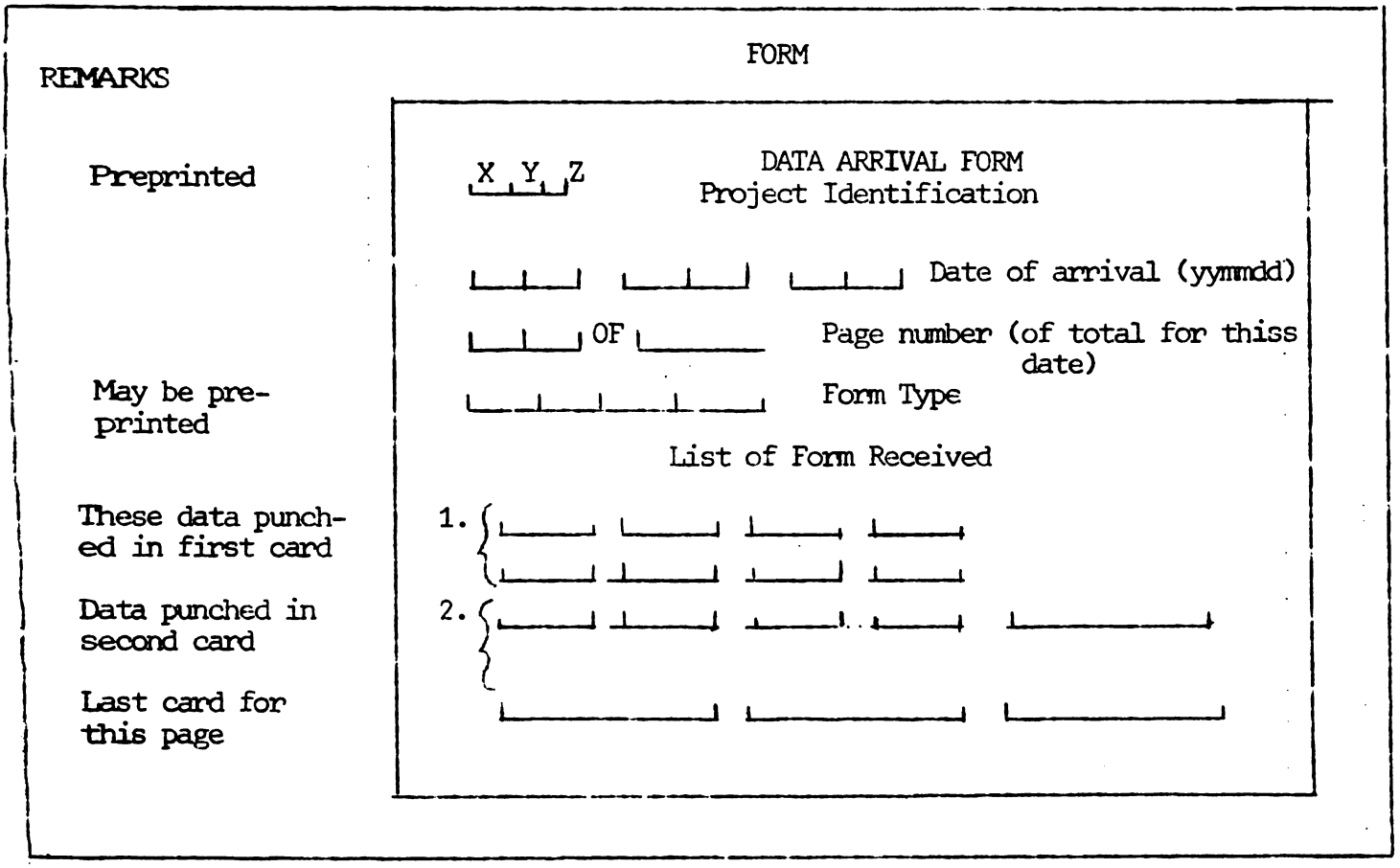
Figure 3.6 FLOWCHART OF THE INVENTORY SUBSYSTEM



formation about the arrival of each data form, but to obtain this information with as little work and cost as possible. Typically, when forms arrive from a remote location, the processing center prepares an 'unpacking form' to record arrival of data. With careful design this form can be the source data form for arrival data (INVCARDS). Figure 3.7 illustrates, schematically, how such a form might appear. Note that several pages of DATA ARRIVAL FORMS may be generated on any one day and each page may generate several cards of information. All the identification information (Project, date of arrival, etc.) is duplicated into each card, of course.

The data fields for a form which has arrived contain the values of all key variables -- enough information to uniquely identify the form and its resulting observations on the MASTER and INVS MASTER files.

Figure 3.7 Schematic Illustration of a Data Arrival Form



The DATA ARRIVAL FORMS are checked for accuracy and punched immediately. (Data forms go through more extensive checking). The resulting cards denoted INVCARDS in the flowchart, are then processed by the INVS Data Input Phase program INVDIP.

11.3 DIP Processing of INVCARDS

Program INVDIP reads the INVCARDS, checks for errors, produces two print files, INVDIP STATUS and INVDIP ERROR MESSAGES, and an output SAS file of inventory transactions INV_TXO.

INVCARDS. The INVCARDS are punched and verified directly from the Data Arrival Form. INVCARDS format obviously depends upon the format of its form.

INVDIP STATUS is a print file containing two sections. The first is a listing of the input data in a format closely resembling the Data Arrival Form. This listing is useful for proofreading input data to check for errors. The second listing is essentially a PROC PRINT listing of the output file, INV_TXO; it is used to check the program's output.

INVDIP ERROR MESSAGES. The INVDIP program performs a variety of error tests of key and data variables, tests for missing cards or 'pages' of input, etc. Any errors or improbable values are indicated in this printout.

The INVDIP program reads the input data, produces the printouts already described, and produces the output file, INV_TXO, sorted by key variables.

INVDIP must also be able to process INVS correction and INVS deletion records, which are used to correct errors that enter the INVS MASTER files. INVS correction and deletion records, and 'new observation' records, contain flags much like the FLAG variable in the DMS correction and deletion observations.

INV_TXO contains one observation for each form received (or error, deletion, or 'new' observation with a FLAG). Each observation contains:

Key variables (as in data MASTERS)

FORM TYPE - (for projects with multiple types of forms)
DATEARR - date of arrival (SAS date variable)
FLAG - to indicate new data, error correction, or deletion record
PAGE - Page number from input data
CARDNO - Card number within page
(other variables on INVS MASTERS).

Note that PAGE and CARDNO are available for checking and are not updated to INVS MASTER.

INVCARDS CORRECTION. After INVDIP is run, the output is carefully checked for error messages and the data printout compared versus the Data Arrival Forms. If errors are found, the INVCARDS are corrected and the job is re-run. No INVS data are moved to the update phase until all errors have been corrected.

This error detection -- correction process has high priority because the INVS files must be updated quickly to serve their purposes.

11.4 INVS File Maintenance Phase

INVS file maintenance is a straightforward SAS file maintenance, based upon the UPDATE statement, with a report generating routine. As noted earlier, INVS update transactions come from several sources, all but one of which are computer-generated. The update program, INVUPDAT, does not distinguish between sources. The lower half of Figure 3.6 has a general flowchart of this phase. The system is described below in terms of its inputs, processing and outputs.

INVENTOR.IMASTERn. The INVS master files have the same key fields as the data master files and, in fact, the latest IMASTER contains:

- One observation for each observation on the data NEWMASTER, plus

- One observation for each observation which has been deleted from the data master files.

The IMASTER data fields are:

- DATEARR -- date the data form arrived
- INPUTDAY - date and time data observation was input by program DIP
- UPDATDAY - date and time the observation was originally added to NEWMASTER by the EDUPDAT program.
- CHECKDAY - Date and time the data observation was most recently checked for data errors by EDUPDAT
- CORRDAY - Date-time the data observation was most recently corrected by EDUPDAT.
- DELDAY -- Date-time the data observation was deleted from the the master file by EDUPDAT. (missing value if data record not deleted).
- ONMASTER - Date and time the data observation was most recently observed by EDUPDATE to be on the data MASTER file. (This should be the date-time of the most recent execution of EDUPDATE unless the observation has been deleted).
- CLEAN - a Boolean variable with value 1 if all variables on the data MASTER observation have status '½' or '¢'. Otherwise CLEAN = 0.

When a new observation is added to IMASTER by an arrival record, most of the fields, representing dates of events which have not yet happened, will have missing values. As transactions come from various stages of processing the missing values. (The missing values will be replaced with real date-time.)

B.INVTXm. The INVS transaction dataset has the same key variables as IMASTER, and the same sort sequence. The other variables in the transaction file depend upon the source of the transaction records. One can review the descriptions of those components to find the detailed descriptions

of INVS transaction files.

INVUPDAT is a straightforward UPDATE program which also checks for INVS transaction data errors, MASTER file update errors, and produces reports on the status of data processing. These reports are described in the following paragraphs. By examining the output reports one can easily determine the structure of the program.

INVUPDAT STATUS REPORTS. INVUPDAT produces several tables of the following general format:

	Arrival	DIP	UPDATE	CHECK	CORP
Arrival	XXX	XXX	XXX	XXX	XXX
DIP		XXX	XXX	XXX	XXX
UPDATE			XXX	XXX	XXX
CHECK				XXX	XXX
CORR					XXX

Several types of entries are made in this type of table. For example, one tabulation is the average "transit times", i.e., time to proceed from step X to step Y. (E.g., in the first column, from Arrival to DIP, Arrival to UPDATE, etc.) Other statistics of interest are the maximum transit times, standard deviation of transit times, etc.

Another type of table is as follows:

Stage	Number of observations		
	Processed	Overdue	Lost
Arrival		---	---
DIP			
Update			
Check			
Corr.			

Here, 'overdue' means that a form has taken too long to progress from one step to another. The actual criteria for 'overdue' status are based on experience with the system.

INVPDAT ALARMS. The INVUPDAT STATUS REPORTS tell project management how many forms have problems; the ALARMS printout lists the identification of the forms which have been determined, in this run, to have problems (be overdue or lost). Typically only one ALARM is issued for each problem which arises. If ALARMS were created for each problem on each update, the data manager would be inundated with redundant ALARMS and they would lose their value.

In addition to these reports, project management may specify other reports to be produced regularly by INVUPDAT. Special purpose reports are also prepared on a regular basis (each three months, for example) or on a special basis, using IMASTER as a data file and the SAS statistical analysis capabilities.

12. The Error Correction Subsystem

The error correction subsystem is basically a human subsystem. Error messages are generated by EDUPDAT. Humans process the messages, determine corrections, key and verify the correction data. The correction DIP program, CORRDIP, is executed to produce a SAS correction transaction dataset and check the correction data for errors. The results printed by CORRDIP are carefully examined to make sure that the corrections do not introduce errors into the data file. If errors are found, the CORRECTION DATA are corrected and CORRDIP is rerun. This process is repeated until it is determined that the correction transactions are entirely correct. Only then is the correction transaction file released to the EDUPDAT program.

As noted, the Error Correction Subsystem is a human system; its design and implementation is a problem in human engineering. The factors which affect its success are the same factors which have a major effect on the success of the entire data management project:

1. Good human engineering
2. Good documentation
3. Good training of operational personnel
4. Good people management
5. Obtaining operational personnel who are good at giving proper attention to detail
6. Having available adequate and appropriate resources: time, space, money, people, hardware, software.

The software problems are not difficult--people problems are crucial. This is true of the operation of the entire data management process.

(Note: This section is incomplete. Additional text will be provided at a later date.)

13. The Quality Control Subsystem

(Note: This section is not yet available; it will be produced at a later date. -- RWH, 08 Feb. 80.)

14. Data Security

(Note: This section is not yet available; it will be produced at a later date --
RWH, 08 Feb. 80)

REFERENCES

BROOKS, F. P. (1975). The Mythical Man-Month, Reading Mass. Addison-Wesley Publishing Company, 1975.

HELMS, R. W. (1972). The Lipids Visit II Data Management System. Documentation at the Central Patient Registry, Lipids Research Clinics Program, Department of Biostatistics. University of North Carolina at Chapel Hill.

HELMS, R. W. (1978). An Overview of Research Data Management with special emphasis on current problems. Proceedings of the ASA Section on Statistical Computing, 1978. The American Statistical Association, Washington, D.C.

HELMS, R. W. (1979), CHRISTIANSEN, D. H., GALLAGHER, P. N. & MORRISSEY, L. J. Designing file structures for longitudinal research data. Proceedings of the American Statistical Computing Section, 1979. The American Statistical Association, Washington, D.C.

METZGER, P. W. (1973). Managing a Programming Project. Englewood Cliffs, N. J. Prentice-Hall, Inc., 1973.

SAS79. SAS User's Guide, 1979 Edition. The SAS Institute Inc., Wesley Publishing Company, 1975.