Sort/Merge with Report Writer User's Manual (AOS and AOS/VS)

093-000155-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000155

Printed in the United States of America
Revision 02, January, 1985

NOTICE

Sort/Merge with
Report Writer
User's Manual
(AOS and AOS/VS)
093-000155-02

Revision History:

Original Release - June, 1978

First Revision - January, 1981

Second Revision - January, 1985

Effective with:

(Sort/Merge with Report Writer (AOS) Rev. 4.10
(Sort/Merge with Report Writer (AOS/VS) Rev. 3.10

CONTENT UNCHANGED

The content and change indicators in this revision are unchanged from 093-000155-01. This revision changes only printing and binding details.

Scientific (S series) ECLIPSE® computers require the optional Character Instruction Set to support any part of the Sort/Merge with Report Writer. In addition, certain functions are not supported on Scientific ECLIPSE® computers even with the Character Instruction Set. These unsupported functions are: (1) sorting and merging on numeric data fields which contain signed binary, decimal or floating point numbers and (2) any use of the Report Writer. (These unsupported functions use the Commercial Instruction Set and therefore are available only on C-, M- or MV- series Commercial ECLIPSE® computers.)

Preface

This manual tells you how to use the AOS Sort/Merge with Report Writer utility (Sort/Merge, for short). It assumes that you are familiar with some of the frequently used CLI commands described in the *Command Line Interpreter User's Manual* (093-000122), and with at least one text editor. A few individual chapters and appendixes assume that you are familiar with other manuals. We tell you what other manuals you need when we summarize the contents of each chapter later in this preface.

How to Read this Manual

Which chapter(s) you read depends on how you plan to use Sort/Merge. Here, we tell you the minimum reading you must do.

If you want just a taste of how Sort/Merge works, read Chapter 2. To get an overview of Sort/Merge's capabilities and the structure of command files (which direct the utility), read Chapters 1 and 3, respectively.

If you want to write a command file that processes AOS sequential files, read at least Chapters 3, 4, and 5. If you want to write a command file that processes INFOS® II files, read at least Chapters 3, 5, and 8.

Read Chapter 2 and/or 7 to learn how to execute a command file.

Clauses and Phrases

We give you the format of each statement that you can use in a command file. When we explain the formats to you, we usually break them down into clauses and phrases. The distinction between the two is simple. A clause contains either IS or ARE; a phrase does not contain IS or ARE. An example of a clause is

```
INPUT FILE IS "name"
```

An example of a phrase is

```
DATA SENSITIVE DELIMITED BY "literal" UPTO integer CHARACTERS
```

Synopsis of Contents

We have arranged this book as follows:

Chapter 1	gives an overview of Sort/Merge's capabilities and introduces the fundamental concept of *key*.
Chapter 2	introduces you to the command file and command line, and gives you a feel for how the utility works. Chapter 2 also elaborates on the concept of key.
Chapter 3	gives an overview of command file structure. We urge you to read this chapter before reading the rest of the manual. It defines terms and explains rules which are stated nowhere else.

Chapter 4	describes command file declarations, except those you'll need for INFOS® II files and Report Writer. The explanation of data types assumes that you are familiar with COBOL data types. If you aren't, read about them in the *COBOL Reference Manual (AOS)* (093-000223).
Chapter 5	describes command file imperatives.
Chapter 6	describes command file massage statements.
Chapter 7	details command lines. If you write command files, we suggest that you read "Detecting Syntax Errors" in the section "Noninteractive Mode." The information there will speed your command file debugging.
Chapter 8	describes INFOS® II input and output file declarations. It assumes that you're already familiar with the INFOS® II system. If you aren't, read the *INFOS® II System User's Manual (AOS)* (093-000152).
Chapter 9	explains how to use Report Writer, and describes the interface between Report Writer and Sort/Merge.
Appendix A	lists all command file statement formats on colored stock.
Appendix B	lists the Sort/Merge error messages with explanations. Sometimes it directs you to explanations in the *AOS Programmer's Manual* (093-000120) or the *INFOS® II System User's Manual.*
Appendix C	lists the Report Writer error messages with explanations.
Appendix D	describes the statistical information Sort/Merge returns after each successful invocation.
Appendix E	advises how you can improve the utility's performance. Read the term definitions in Appendix D before reading this appendix.
Appendix F	is the ASCII character code set.

What Do You Think?

At the end of this manual you'll find a Comments Form. This is your direct line to us at Software Documentation--please use it. We want to know what you like and dislike about the manual. We welcome your suggestions, and we really listen.

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	**Means**
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use:

$$\left\{\begin{matrix}\text{required}_1\\ \text{required}_2\end{matrix}\right\}$$

which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.

[optional] You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.

... You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
⤶	Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., < 012 > represents octal 12.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY⤶
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the CLI prompt.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.

End of Preface

Contents

Chapter 1 - Introduction

Chapter 2 - Sort and Merge Examples

Chapter 3 - Command File Overview

Chapter 4 - Declarations

Chapter 5 - Imperatives

Chapter 6 - Massage Statements

Chapter 7 - Command Lines

Chapter 8 - INFOS II Files

Chapter 9 - Report Writer

Appendix A - Command Line and Command File Statement Summary

Appendix B - Error Messages

Appendix C - RWCHECK Error Messages

Tables

Illustrations

Chapter 1

Introduction

Data General's AOS Sort/Merge is a general-purpose utility which manipulates record order and content. It runs under the Advanced Operating System (AOS) and runs as a 16-bit subsystem under the Advanced Operating System/Virtual Storage (AOS/VS).

Sort/Merge gives you the power to

- sort and copy records
- merge multiple files into a single file
- edit record fields
- delete duplicate output records from a sort or merge operation
- delete records according to conditions that you specify. You can also write these deleted records to other files
- neatly format output records into reports with the report writer feature

To take full advantage of the utility's many features, you use a *command file,* which contains statements described in this manual. Since you can save a command file on disk, you need write a command file only once to perform repetitive tasks.

The rest of this chapter describes Sort/Merge's features in more detail.

Records and Files

Sort/Merge accepts any AOS-generated fixed-length, variable-length or data-sensitive (text) file for input or output. It also accepts any INFOS® II-generated indexed sequential (ISAM) or database (DBAM) file for input or output. Some of the high-level languages that generate these files are Fortran IV, Fortran 5, PL/I, DG/L ™, RPG II, ANSI'74 COBOL, Extended BASIC, and Idea software.

File size is limited only by the amount of available disk storage. Record size is limited by your choice of processing options; in any case, you can always use at least 3,000-character records.

Keys

You define one or more keys in a command file. The keys tell Sort/Merge the criteria by which you want to order your output records. For example, if you want to sort input records by name, you can make the records' surname field the primary key, their first name field the secondary key, and their middle initial field the tertiary key. Sort/Merge will then order the records in this order: first, by last name; second, by first name; and third, by middle initial. The order in which you define keys is the order in which the utility sorts or merges records. You can specify either an ascending or a descending ASCII sequence for the keys. You can even specify an ascending sequence for one key and a descending sequence for another.

The utility accepts a variety of key types. The default key type is essentially the same as a COBOL alphanumeric data type; we call it a *character* key type. You have the option of specifying any of the COBOL data types as your key type. And if you specify a *decimal* key type, you can also specify leading or trailing overpunch and/or leading or trailing signs.

Sorting

There are four basic sort operations you can choose from: sort, stable sort, tag sort, and stable tag sort. The sort and stable sort both carry the entire record throughout all phases of the utility's operation. However, the stable sort works differently than the sort. It guarantees that input records whose sort keys are exact duplicates are written to the output file in exactly the order that the utility encounters them. The tag sort creates a tag identifying each input record and carries the tags through to the output phase. During the output phase, the utility retrieves the records and writes them to the output file. Because the utility isn't carrying the entire record through every tag sort phase, a tag sort requires less disk space for its execution. The difference between a stable tag sort and a tag sort is the same as the difference between a sort and a stable sort.

You can request the utility to delete duplicate output records with each of the four sorts.

Merging and Copying

The utility merges files by collating the contents of two or more sorted files into a single output file.

Sort/Merge copies the contents of one or more input files to a single output file, without changing record order.

Massaging Records

We call the process of editing records *massaging*. You can massage input and output file records that are sorted, merged, or copied.

The following is an overview of the utility's varied massaging features. You can

- rearrange, duplicate, and delete fields in a record
- replace a field with a smaller, equally-sized, or larger field
- insert a field and a record number in a record
- convert variable-length records to fixed-length by padding them with an ASCII character
- translate an entire record or selected fields of a record from ASCII to EBCDIC or from EBCDIC to ASCII. Or, you can define a translation table to perform any desired translation. Also, you can translate lowercase ASCII letters to uppercase
- exclude certain characters when Sort/Merge determines a record's position in the output. For example, your key field may contain a special character, like /. If you don't want the utility to consider the / when determining a record's output position, you can compress the key field to remove the / during input and reinsert it during output
- skip a record if a condition is met. Skipping means that you don't want the utility to write certain records to the output file. You can save skipped records in one or more secondary output files, called *skip files*. Thus, you can split an input file into several output files

Altering the Collating Sequence

Normally, the utility collates your output in ascending sequence according to ASCII character values. However, for each sort or merge key you define, you can specify either an ascending or descending collating sequence. Optionally, you can define a table in which you assign alternative collating values to characters, and then tell the utility to order your output using that table.

Report Writer

Sort/Merge uses a Report Writer to produce reports. Although Sort/Merge's massage features can give an output file a neat appearance and useful organization, Report Writer can further improve an output file's appearance and usefulness. For example, Report Writer can insert column headings and perform arithmetic calculations of variables. You might, for instance, want the total value of all inventory items subtotaled and listed under the column heading "Stock Value." Report Writer can calculate this information.

End of Chapter

Chapter 2

Sort and Merge Examples

This chapter leads you through examples of sort and merge processing. The first two sections explain information you will need to understand the examples.

Command Files

A Sort/Merge *command file* contains statements that tell the utility

- where to find the records to sort or merge
- where to send the sorted or merged records
- how to perform the sort or merge

We pattern command files in this chapter's examples after the following command file structure:

```
INPUT FILE declaration(s).
OUTPUT FILE declaration.
KEY declaration(s).
SORT or MERGE imperative.
END statement.
```

You must end each statement with a period.

The examples illustrate the following functions of these command file statements:

Statement	Function
INPUT FILE declaration	specifies the name of an input file and its record type
OUTPUT FILE declaration	specifies the name of the output file
KEY declaration	defines the *key* characters or groups of characters on which the utility bases the sort or merge
SORT or MERGE imperative	tells the utility to perform a sort or merge process
END statement	signals the end of the command file

You create a command file as you would any other AOS file: with a text editor or the AOS CLI CREATE command.

Command Lines

The *command line* invokes the utility and indicates the name of the command file you want to use. Chapter 7 describes its other purposes.

The command line format we use for the examples in this chapter is

{SORT / MERGE} /C=filename *[/O]*

/C=filename	is a command line *switch* identifying the command file's name, filename.
/O	is a command line switch you must include when using an existing file as the output file.

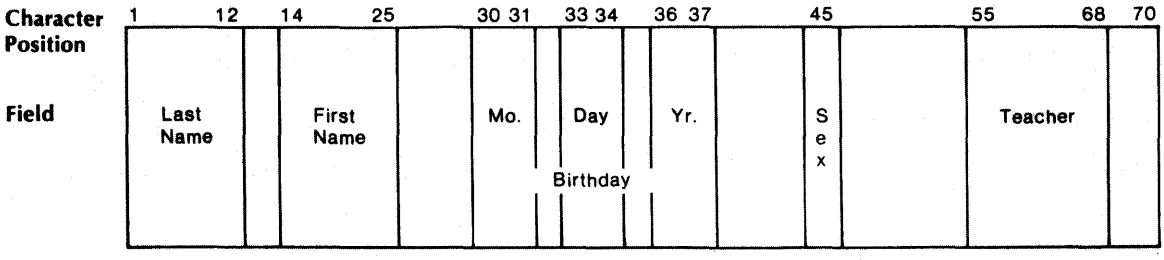
For example, if you use a command file named PART_SORT to sort records into an existing output file, the command line would be

```
)SORT/C=PART_SORT/O
```

(The utility deletes the existing output file and then recreates it using the sorted records.) After you type the command line, the utility performs the desired process and returns statistical information which we discuss in Appendix D.

Examples

All the examples in this chapter use 70-character fixed-length records. Figure 2-1 shows the records' format; the format shows which character positions each field occupies.



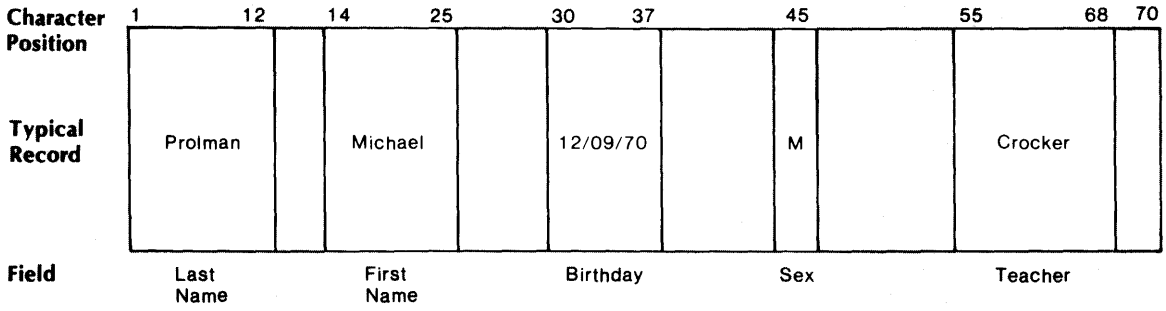
SD-02348

Figure 2-1. Record Format for Examples

A typical record looks like this:

PROLMAN MICHAEL 12/09/70 M CROCKER

Figure 2-2 superimposes the typical record over the record format to further clarify the relationship between the two.



SD-02349

Figure 2-2. Typical Record Superimposed over Format

If you try these examples at your terminal, type NEW LINE as the 70th character.

Each example discusses

1. the task you want to accomplish
2. the input file(s) before the sort or merge
3. the command file(s) needed to accomplish the task
4. the command line(s) needed with the command file(s)
5. the output file after the sort or merge

Last Name Sort

You want to sort a sixth grade class register file named REGISTER_6, based on the last name field. You need a command file which sorts the records in REGISTER_6, the input file. You also need a command line and an output file. REGISTER_6 already exists and contains these records:

PROLMAN	MICHAEL	12/09/70	M	CROCKER
POWERS	GREGORY	05/19/70	M	LEVITT
PUZNIAK	KAREN	03/19/70	F	LEVITT
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PRATT	JULIA	10/21/70	F	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT
PROUT	JANET	07/21/70	F	CROCKER
PRESTON	TONY	06/06/70	M	LEVITT
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK

Create the command file LAST_NAME_SORT to contain these statements:

```
INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "REGISTER_OUT".
KEY 1/12.
SORT.
END.
```

The RECORDS ARE clause of the INPUT FILE declaration names an input file with fixed-length records of 70 characters.

The KEY declaration instructs the utility to base the sort on characters 1 through 12 only. By default, the KEY declaration instructs the utility to sort the records in ascending order.

After you create the command file, execute the following command line. It invokes the utility and indicates the name of the command file:

```
)SORT/C=LAST_NAME_SORT/O)
```

You use the /O switch to indicate that the output file REGISTER_OUT already exists.

After you execute the command line, the output file contains the same REGISTER_6 records, in a new order:

POWERS	GREGORY	05/19/70	M	LEVITT
PRATT	JULIA	10/21/70	F	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROUT	JANET	07/21/70	F	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PUZNIAK	KAREN	03/19/70	F	LEVITT

Teacher Sort

You want to sort REGISTER_6 based on the teacher field. The input file REGISTER_6 contains

PROLMAN	MICHAEL	12/09/70	M	CROCKER
POWERS	GREGORY	05/19/70	M	LEVITT
PUZNIAK	KAREN	03/19/70	F	LEVITT
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PRATT	JULIA	10/21/70	F	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT
PROUT	JANET	07/21/70	F	CROCKER
PRESTON	TONY	06/06/70	M	LEVITT
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK

The command file TEACHER_SORT contains

```
INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "TEACHERS".
KEY 55/68.
SORT.
END.
```

The KEY declaration instructs the utility to base the sort on characters 55 through 68 only.

The following command line tells the utility to sort records into output file TEACHERS, which already exists:

```
)SORT/C=TEACHER_SORT/O)
```

After you execute the command line, the output file contains

PROUT	JANET	07/21/70	F	CROCKER
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRATT	JULIA	10/21/70	F	CROCKER
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PUZNIAK	KAREN	03/19/70	F	LEVITT
POWERS	GREGORY	05/19/70	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PRENDERGAST	JOHN	01/05/71	M	LEVITT

Birthday Sort

You want to sort REGISTER_6 based on the birthday field. The input file REGISTER_6 contains

PROLMAN	MICHAEL	12/09/70	M	CROCKER
POWERS	GREGORY	05/19/70	M	LEVITT
PUZNIAK	KAREN	03/19/70	F	LEVITT
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PRATT	JULIA	10/21/70	F	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT
PROUT	JANET	07/21/70	F	CROCKER
PRESTON	TONY	06/06/70	M	LEVITT
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK

This example raises an important question: how do you resolve key conflicts? For example, consider the Prinsky, Provenzani, and Powers records. If you make the birthday field the key, then the utility will put Provenzani (05/12/70) before Prinsky (12/28/69). Obviously, a student born in 1969 belongs before a student born in 1970, even if the 1970 student was born in May and the 1969 student in December. How do you resolve this conflict?

You divide the birthday field into two subfields by using two KEY declarations: the first declares the year subfield and the second declares the month subfield. The utility makes the year key the primary key and the month key the secondary key. In this way, the utility sorts the birthdays by year first and by month second. Figure 2-3 shows the birthday field divided into two subfields.

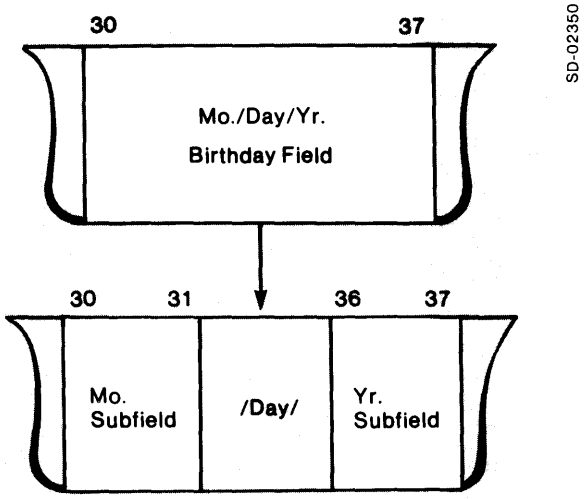


Figure 2-3. Birthday Field Divided into Two Subfields

Now compare the Provenzani and Powers records. In our current scheme, Sort/Merge compares both records' year and month subfields and finds them the same. How do you make the utility resolve this conflict? You divide the birthday field still further into three subfields; use a third KEY declaration for the day subfield. The year and month keys remain the primary and secondary keys, but the day key becomes the tertiary key. This third key makes the utility place Provenzani (05/12/70) before Powers (05/19/70). Figure 2-4 shows the birthday field divided into three subfields.

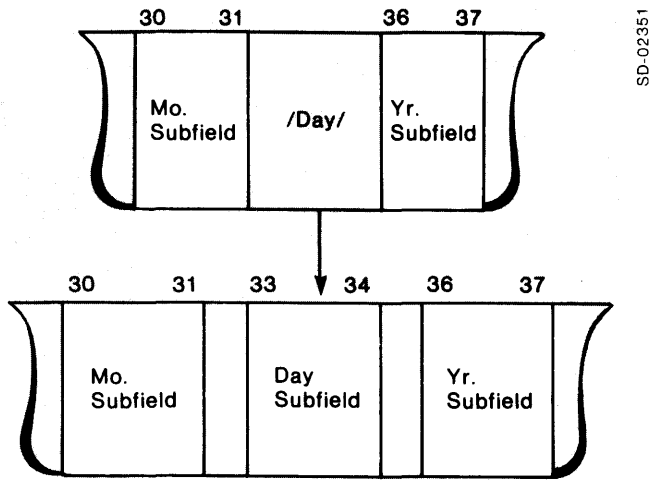


Figure 2-4. Birthday Field Divided into Three Subfields

We use multiple keys in the command file BIRTHDAY_SORT, which contains

```
INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "BIRTHDAY".
KEY 36/37.
KEY 30/31.
KEY 33/34.
SORT.
END.
```

Let's summarize a use of multiple keys in general and in this example. You can include more than one KEY declaration in a command file. The utility first bases the sort on the KEY declaration appearing first in the command file. This declaration is the primary key. To resolve conflicts between records, the utility will use the KEY declaration appearing second in the command file, if you included one. The second KEY declaration is the secondary key. To resolve further conflicts between records, the utility can use a third KEY declaration, the tertiary key, and so on. In this example, we sorted the birthdays by year, then month, and then day, by declaring a KEY declaration for the year subfield, a second KEY declaration for the month subfield, and a third KEY declaration for the day subfield. Put another way, we sorted the birthdays by year, then month, then day, by declaring a year key, then a month key, then a day key.

The command line you need is

```
)SORT/C=BIRTHDAY_SORT)
```

The /O switch does not appear in the command line because the output file BIRTHDAY does not yet exist. The utility will create it.

After the utility creates the ouput file, it contains

PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PUZNIAK	KAREN	03/19/70	F	LEVITT
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
POWERS	GREGORY	05/19/70	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PROUT	JANET	07/21/70	F	CROCKER
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PRATT	JULIA	10/21/70	F	CROCKER
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT

New Student Merge

You want to merge file NEW_STUDENTS with TEACHERS, based on the teacher field. The input file TEACHERS contains

PROUT	JANET	07/21/70	F	CROCKER
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRATT	JULIA	10/21/70	F	CROCKER
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PUZNIAK	KAREN	03/19/70	F	LEVITT
POWERS	GREGORY	05/19/70	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PRENDERGAST	JOHN	01/05/71	M	LEVITT

The second input file NEW_STUDENTS contains

PATTERSON	DAVID	01/01/70	M	KIRKPATRICK
PARK	DONA	11/03/70	F	LEVITT
PARASKEVAS	WILLIAM	09/12/70	M	CROCKER

The merge process requires previously sorted input files. The command file TEACHER_SORT already prepared TEACHERS for merging. TEACHER_SORT.FOR.NEW_STUDENTS, the command file which prepares NEW_STUDENTS for merging, contains these statements:

```
INPUT FILE IS "NEW_STUDENTS", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "NEW_STUDENTS_OUT".
KEY 55/68.
SORT.
END.
```

After the preparatory sorts, TEACHERS remains the same as above, and NEW_STUDENTS_OUT contains

PARASKEVAS	WILLIAM	09/12/70	M	CROCKER
PATTERSON	DAVID	01/01/70	M	KIRKPATRICK
PARK	DONA	11/03/70	F	LEVITT

The following command file, NEW_STUDENTS.WITH.TEACHERS, merges the two sorted input files:

```
INPUT FILE IS "NEW_STUDENTS_OUT", RECORDS ARE 70 CHARACTERS.
INPUT FILE IS "TEACHERS", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "MASTER_6".
KEY 55/68.
MERGE.
END.
```

You need three command lines to perform this merge. In the teacher sort example, you already prepared TEACHERS for merging by executing this command line:

```
)SORT/C=TEACHER_SORT/O)
```

To sort NEW_STUDENTS and thus prepare NEW_STUDENTS_OUT as an input file for merging, execute this command line:

```
)SORT/C=TEACHER_SORT.FOR.NEW_STUDENTS/O)
```

Finally, to merge the two input files TEACHERS and NEW_STUDENTS_OUT into output file MASTER_6, execute this command line:

```
)MERGE/C=NEW_STUDENTS.WITH.TEACHERS/O)
```

After executing these command lines, the output file MASTER_6 contains

PARASKEVAS	WILLIAM	09/12/70	M	CROCKER
PROUT	JANET	07/21/70	F	CROCKER
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRATT	JULIA	10/21/70	F	CROCKER
PATTERSON	DAVID	01/01/70	M	KIRKPATRICK
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
PARK	DONA	11/03/70	F	LEVITT
PUZNIAK	KAREN	03/19/70	F	LEVITT
POWERS	GREGORY	05/19/70	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PRENDERGAST	JOHN	01/05/71	M	LEVITT

Male and Female Student Sort

Input file MALES contains records for male students and input file FEMALES contains records for female students. Merge MALES with FEMALES based first on the teacher field, then on the last name field, and finally on the first name field. Use the SORT imperative and one command file to perform the equivalent of both the merge and the preparatory sorts.

Input file MALES contains

PROVENZANI	ANTHONY	05/12/70	M	CROCKER
POWERS	GREGORY	05/19/70	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROLMAN	GERALD	12/09/70	M	CROCKER
PRENDERGAST	JOHN	01/05/71	M	LEVITT

Input file FEMALES contains

PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PRATT	JULIA	10/21/70	F	CROCKER
PROLMAN	MAXINE	12/09/70	F	CROCKER
PUZNIAK	KAREN	03/19/70	F	LEVITT
PROUT	JANET	07/21/70	F	CROCKER
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK

You use the command file TEACHER_AND_WHOLE_NAME which contains

```
INPUT FILE IS "MALES", RECORDS ARE 70 CHARACTERS.
INPUT FILE IS "FEMALES", RECORDS ARE 70 CHARACTERS.
OUTPUT FILE IS "MASTER_6".
KEY 55/68.
KEY 1/12.
KEY 14/25.
SORT.
END.
```

As the command file indicates, you can declare more than one input file for a sort process. The utility reads each input file in the order that you specify them, in effect creating one large input file for the sort.

The KEY declarations specify the teacher field as the primary key, the last name field as the secondary key, and the first name field as the tertiary key. The command line is

```
)SORT/C=TEACHER_AND_WHOLE_NAME)
```

The output file MASTER_6 contains

PRATT	JULIA	10/21/70	F	CROCKER
PROLMAN	GERALD	12/09/70	M	CROCKER
PROLMAN	MAXINE	12/09/70	F	CROCKER
PROLMAN	MICHAEL	12/09/70	M	CROCKER
PROUT	JANET	07/21/70	F	CROCKER
PROVENZANI	ANTHONY	05/12/70	M	CROCKER
PRINSKY	SUSAN	12/28/69	F	KIRKPATRICK
PUTNAM	GAIL	08/25/70	F	KIRKPATRICK
POWERS	GREGORY	05/19/70	M	LEVITT
PRENDERGAST	JOHN	01/05/71	M	LEVITT
PRESTON	TONY	06/06/70	M	LEVITT
PUZNIAK	KAREN	03/19/70	F	LEVITT

End of Chapter

Chapter 3

Command File Overview

This chapter defines terms and explains rules that we use throughout the rest of the manual. Also, it presents an overview of command files.

Terms You'll Need to Know

Throughout the rest of this manual you'll find AOS and Sort/Merge terms. We define these terms here.

Record Types

data-sensitive	Records delimited by a particular character. We'll refer to data-sensitive records delimited by any one of the default delimiters -- NEW LINE, null, or form feed -- as *standard* data-sensitive records, and those delimited by characters other than the default delimiters as *nonstandard* data-sensitive records. Nonstandard also refers to data-sensitive records which exceed 136 characters. In both data-sensitive types, the delimiter is part of the record. Both types can have constant or varied length.
fixed-length	Records whose lengths are constant.
variable-length	Records whose lengths vary. The records contain a header that tells you the record length.
dynamic	Records whose lengths you specify when you read or write them.

NOTE: Sort/Merge does not recognize dynamic records. To process these with Sort/Merge, define them as one of the other record types in your input or output file declarations. (See Chapter 4.)

Storage Terms

block size	The number of bytes (characters) in a physical tape block. AOS default block size is 2048 bytes.
element size	The number of sectors in each element that AOS allocates for a disk file. Elements are blocks of sectors (512-byte data blocks).

Sort/Merge Terms

literal	A string of one or more ASCII characters.
collating value	A character's value in a character code set (often ASCII). Sort/Merge orders characters based on their collating values. Appendix C lists the ASCII and EBCDIC collating values. For example, < 040 > is the ASCII collating value of a blank space. Sort/Merge allows you to alter a character's collating value.
collating sequence	The order of a set of characters based on its collating values. By altering collating values, you can alter a collating sequence.
location phrase	The notation used to indicate a single character or range of characters in a record. For example, 2/2 indicates the second character position in a record; 2/4 indicates the second through fourth character positions.
Sort/Merge process	The steps that Sort/Merge takes to read, order, edit, and write records.

Command File Overview

A command file is a series of statements. Different types of statements serve different functions:

This type of statement	tells the Sort/Merge utility
declaration	the names of the input and output files and their record type
	the field(s) on which to base the Sort/Merge process
	the collating values of the records' characters
	the temporary files used for intermediate storage of records during a Sort/Merge process
massage	how to edit the input and/or output records
	which records to use in the Sort/Merge process
imperative	which Sort/Merge process to use
END	where the command file ends

Table 3-1 shows you the structure of a command file, the types of command file statements, the required and optional statements, and the number of each statement allowed in one command file. The word "many" appears under the column head "Number Allowed." In general, "many" means as many as you want. Only memory space can limit the number of statements where many are allowed.

Table 3-1. Overview of Command File Structure *

Type	Statement	Required/ Optional	Number Allowed
Declaration	INPUT FILE	Required for non-INFOS II input file	Many
	INPUT INFOS	Required for INFOS II input file	Many
	OUTPUT FILE	Required for non-INFOS II output file	One
	OUTPUT INFOS	Required for INFOS II output file	One
	OUTPUT REPORT	Optional	One
	TABLE	Optional	Many
	KEY	Optional	Many
	WORK FILE	Optional	Many
Massage (for input records)	COMPRESS IF INSERT (including INSERT TAG) PAD REFORMAT REPLACE REPLACE TABS TRANSLATE	Optional	Many of each
Imperative	SORT STABLE SORT TAG SORT STABLE TAG SORT DELETING DUPLICATES MERGE COPY	Required	Only one of one
Massage (for output records)	Same statements as for input records, except INSERT TAG	Optional	Many of each
END	END	Required	One

* This table assumes that you use the noninteractive mode command line (discussed in Chapter 7) to invoke the command file.

Certain rules govern statement order in command files:

- You must place declarations before all other types of statements. Within the declaration section, the input file declarations come first, followed by (in order): one output file declaration, TABLE declarations (if any), KEY declarations (if any), and WORK FILE declarations (if any).

  OUTPUT FILE, OUTPUT INFOS, and OUTPUT REPORT are all output file declarations. In any command file, you may specify only one of them.

- You can place massage statements before and/or after the imperative. Massage statements for input records precede the imperative; massage statements for output records follow the imperative.

- Place the END statement last in the command file.

- You must terminate each statement with a period.

See Chapter 8 for information about the INPUT INFOS and OUTPUT INFOS declarations.

You can arrange statements on a page in any way you like, as long as they are syntactically correct. A declaration or massage statement can span several lines. Also, spaces, tabs, form feeds, and NEW LINEs can separate declaration phrases and massage statements.

Comment Feature

You can place comments on any line in the command file; the utility will not try to execute them. The comment format is very simple:

```
% comment
```

You must place % in column 1.

For example:

```
% This command file sorts my address list by zip code.
INPUT FILE IS "ADDRESS_LIST_BY_NAME", RECORDS ARE 87 CHARACTERS.
OUTPUT FILE IS "ADDRESS_LIST_BY_ZIP".
% The zip code field is character positions 83 through 87.
KEY 83/87.
SORT.
END.
```

Statement Formats: General Information

This section explains the syntax you'll find common to most command file statements.

Location Phrase

The location phrase restricts the character positions that the utility uses in a Sort/Merge process. For example, in the teacher sort example of Chapter 2, we restricted the sort key to the character positions containing the teacher field. The format for the location phrase is

```
integer/integer
integer/LAST
```

The integer before the / is the first character of the range; the integer after the / is the last, and must equal or exceed the integer before the /. LAST is the last character of a given record. Thus, you must use the integer/LAST form when you refer to the last character of records whose lengths vary.

Every character position in a record matters, even if it contains a TAB, blank, form feed, null, or NEW LINE character. The positions are consecutively numbered; the first is numbered one.

Let's take some examples. If a record is 80 characters long, then the location phrase for a five character field starting with character position 50 is 50/54. The fiftieth character is 50/50. The location phrase for the entire record is 1/80 or 1/LAST. Specify the last 30 characters of the record as 51/80 or 51/LAST.

Literals

A literal is a string of characters. Delimit literals with either apostrophes or double quotation marks. Do not mix delimter characters. For example, the ASCII character A is correctly delimited by "A" or 'A', but not "A'. Within a literal, you can represent single characters by the character's octal value enclosed in angle brackets < > . For example, all of the following represent the single-character literal ASCII A:

"A" or 'A' or " < 101 > " or ' < 101 > '

What if you want to use a literal delimiter (" or ') as a literal? To represent " as a literal, use one of these:

' " ' or " < 042 > "

To represent ' as a literal, use one of these:

" ' " or " < 047 > "

Single Character

The format for a single character is

{ integer }
{ "literal" }

The integer represents the character's corresponding ASCII decimal value. For example, the decimal value of an ASCII $ is 36. Thus, you can specify the ASCII character $ as 36 or "$".

Abbreviations

The Sort/Merge utility accepts one set of abbreviations for five words in command file statements.

For	you can use
CHARACTERS	CHARS
RECORDS	RECS
DATA SENSITIVE	DATA SENS
ASCENDING	ASC
DESCENDING	DESC

End of Chapter

Chapter 4

Declarations

This chapter describes all command file declarations except the INPUT INFOS and OUTPUT INFOS declarations (see Chapter 8), and the OUTPUT REPORT declaration (see Chapter 9).

AOS INPUT FILE Declaration

The AOS INPUT FILE declaration names an input file and, optionally, describes its record type and block size.

The format of an AOS INPUT FILE declaration is

```
INPUT FILE IS "name"

[ ,RECORDS ARE { integer CHARACTERS                                                    }
               { DATA SENSITIVE [DELIMITED BY "literal"] UPTO integer CHARACTERS       }
               { VARIABLE UPTO integer CHARACTERS                                      } ]

[, BLOCKS ARE integer CHARACTERS ] .
```

You may omit the RECORDS ARE clause only if the input file consists of fixed-length or standard data-sensitive records. If you are not sure of a file's record type, enter FILESTATUS/RECORD from the appropriate directory. If the records are dynamic or are not a specific record type, you may treat them as

- data-sensitive if each record ends with a data-sensitive delimiter (default or user defined)
- fixed-length if all the records are the same length
- variable-length if each record contains a header consistent with the AOS variable-length record format (indicating the record length)

integer CHARACTERS Phrase

You may use the integer CHARACTERS phrase for an input file consisting of fixed-length records. You're required to use the phrase if the input file contains fixed-length records, but the FILESTATUS command doesn't indicate that the file has a fixed-length format. The argument integer is the record length in characters. For example, if input file records are all 80 characters long, then use this phrase:

```
RECORDS ARE 80 CHARACTERS.
```

DATA SENSITIVE Phrase

You must use the DATA SENSITIVE phrase for an input file consisting of nonstandard data-sensitive records. The phrase is optional for an input file consisting of standard data-sensitive records.

The longest record in the input file determines the integer in the UPTO integer phrase. If you know the length in characters of the longest record, specify that as integer . If you don't know the length of the longest record, you must determine a length which you think no record will exceed; use that length as integer. Try for a tight fit. That is, choose a number high enough for, but as close as possible to, the length of the longest record. For example, if you think that the largest record in a file is about 190 characters, you could let integer equal 250 to be very safe. But 200 is probably a better choice. If you choose 200, use this phrase:

```
UPTO 200 CHARACTERS.
```

The DELIMITED BY phrase lets you select a delimiter other than the AOS default delimiters (NEW LINE, null, and form feed). Each character in literal indicates a record delimiter. For example,

```
DELIMITED BY "/*;"
```

tells the utility to delimit a record whenever it finds a slash, asterisk, or a semicolon. Note that the characters in literal are the only delimiters. Thus, if you want both the AOS default delimiters and your own, you must explicity list them all. For example, to use NEW LINE and * as delimiters, use this phrase:

```
DELIMITED BY "<012>*"
```

VARIABLE UPTO Phrase

You must use the VARIABLE UPTO phrase if the input file consists of variable-length records. If you know the length of the longest record, specify that as integer . Otherwise, choose integer so that you will safely include the longest record in the sort or merge. But also try for a tight fit, the same way that you would for a file of data-sensitive records whose maximum length is uncertain. For example, if you think that your longest record is about 75 characters, you could specify 85 characters:

```
VARIABLE UPTO 85 CHARACTERS
```

BLOCKS ARE Clause

The BLOCKS ARE clause states the number of characters (bytes) in a physical block of a magnetic tape file. Use this clause if the tape that you're reading doesn't have the default AOS blocksize of 2,048 characters. If the size of the blocks on the tape varies, you must specify integer to be at least as long as the longest block in the file. If you don't, blocks longer than the size that you specify are truncated.

Examples

Input file FILE_ONE has fixed-length records 40 characters long. The declaration for this file is either

```
INPUT FILE IS "FILE_ONE", RECORDS ARE 40 CHARACTERS.
```

or

```
INPUT FILE IS "FILE_ONE".
```

FILE_TWO has data-sensitive records no longer than 150 characters delimited by *, /, or $. The declaration for this file is

```
INPUT  FILE IS "FILE_TWO",
       RECORDS ARE DATA SENSITIVE
       DELIMITED BY "*/$"
       UPTO 150 CHARACTERS.
```

An input tape file has fixed-length records 100 characters long and a block size of 2,000. A declaration for this file is

```
INPUT FILE IS "@MTA2:3",
   RECORDS ARE 100 CHARACTERS, BLOCKS ARE 2,000 CHARACTERS.
```

Notice that we formatted each declaration a different way. You can choose any formatting scheme you like.

AOS OUTPUT FILE Declaration

The AOS OUTPUT FILE declaration names an output file. Optionally, it describes the file's record type, block size, and element size.

The format of an AOS OUTPUT FILE declaration is

OUTPUT FILE IS "name"

$$\left[\begin{array}{l} ,RECORDS\ ARE \left\{\begin{array}{l} integer\ CHARACTERS \\ DATA\ SENSITIVE\ [DELIMITED\ BY\ "literal"]\ UPTO\ integer\ CHARACTERS \\ VARIABLE\ UPTO\ integer\ CHARACTERS \end{array}\right\} \\ \left[\left\{\begin{array}{l} BLOCKS\ ARE\ integer\ CHARACTERS \\ ,ELEMENTS\ ARE\ integer\ BLOCKS \end{array}\right\}\right] \end{array}\right] .$$

The RECORDS ARE clause works exactly like the one for input files. (See "AOS INPUT FILE Declaration" in this manual.)

BLOCKS ARE Clause

The BLOCKS ARE clause states the number of characters (bytes) in a physical block of a magnetic tape file. Use this clause if the file that you're writing to doesn't have the default AOS blocksize of 2,048 characters.

Specifying a large block size reduces I/O time and saves tape.

ELEMENTS ARE Clause

The ELEMENTS ARE clause describes the file's size in physical units on the disk. The default element size is one (512-bytes). See "Increase Element Size" in Appendix E for an explanation of why you would want to specify the element size.

TABLE Declaration

The TABLE declaration lets you change the collating values of characters. Since a collating sequence depends on collating values, changing collating values changes a collating sequence. For example, the ASCII decimal equivalent of 9 is 57 and A is 65. Consequently, for an ascending sequence, the utility will output records whose key fields begin with 9 before records whose key fields begin with A. Suppose that the utility sorts an inventory list in ascending order, based on a five character inventory code. A section of the output file might look like this:

9A542	desk	2 drawer	brown
9A544	desk	4 drawer	beige
A3217	pen	felt tip	blue
A3221	pen	felt tip	red

Note that the records with inventory codes starting with 9 come before those starting with A. If we reverse the collating values of 9 and A, then the collating sequence of the inventory records changes:

A3217	pen	felt tip	blue
A3221	pen	felt tip	red
9A542	desk	2 drawer	brown
9A544	desk	4 drawer	beige

You'll need a TABLE declaration when you want to

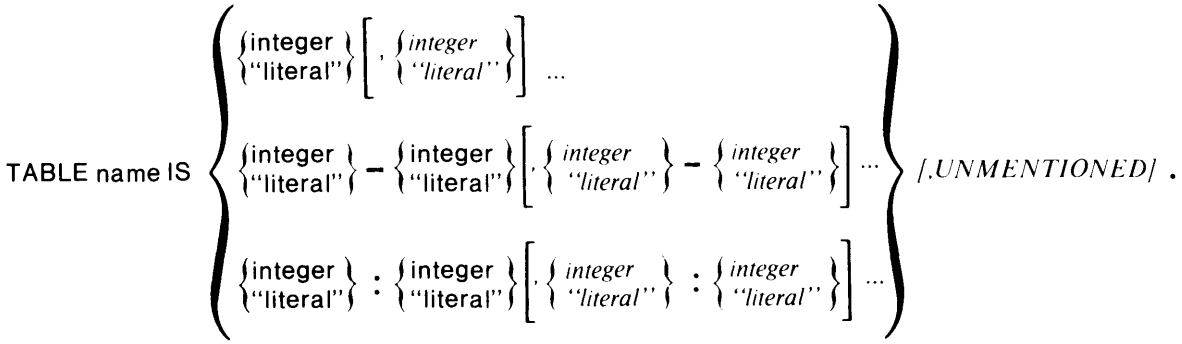
- change the normal collating sequence of your output
- ignore one or more characters in a record by using the COMPRESS massage statement
- translate data other than EBCDIC to ASCII, or ASCII to EBCDIC, or lowercase ASCII to uppercase ASCII

You don't need a TABLE declaration when you want to

- change the collating sequence from ascending to descending; instead, use DESCENDING in the KEY declaration, described in the next section
- translate data written in ASCII to EBCDIC, or vice versa, because the utility provides predefined tables for you
- translate lowercase ASCII to uppercase ASCII because the utility provides a predefined table for this purpose

These are the formats for a TABLE declaration:

FORMAT ONE



FORMAT TWO

```
                  ⎧ ASCII           ⎫
                  ⎪ ASCII_TO_EBCDIC ⎪
TABLE name FROM   ⎨ EBCDIC_TO_ASCII ⎬ IS "literal"=integer [,"literal"=integer] ... .
                  ⎪ LOWER_TO_UPPER  ⎪
                  ⎩ name            ⎭
```

FORMAT THREE

```
TABLE name1 IS FILE "name2 ".
```

In all three formats, the **name** argument in **TABLE name** must consist only of uppercase letters, digits, and the underline character. Also, this **name** is not a literal, so don't delimit it with ' or ".

Notice that format one has three lines between the two main set of braces. We refer to the top line as option one, the middle line as option two, and the third line as option three.

Table 4-1 is a subset of the ASCII character set (shown fully in Apendix F). You can refer to Table 4-1 for most of the examples in this section. Notice that decimals 65 through 90 represent *uppercase* letters. The lowercase letters have different ASCII decimal equivalents. So, for example, don't confuse the uppercase letter J with the lowercase letter j. Also, don't confuse the digit 0 with the lowercase letter o or the uppercase letter O.

Table 4-1. Subset of the ASCII Character Set

ASCII Character	ASCII Decimal Equivalent	ASCII Character	ASCII Decimal Equivalent
0	48	I	73
1	49	J	74
2	50	K	75
3	51	L	76
4	52	M	77
5	53	N	78
6	54	O	79
7	55	P	80
8	56	Q	81
9	57	R	82
A	65	S	83
B	66	T	84
C	67	U	85
D	68	V	86
E	69	W	87
F	70	X	88
G	71	Y	89
H	72	Z	90

Format One

A list of characters follows TABLE name IS. You can represent a character either as a literal or as a decimal number.

The TABLE name IS phrase's option one is

TABLE name IS { integer / "literal" } [, { *integer* / *"literal"* }] ...

literal is one or more ASCII characters and integer is the decimal equivalent of an ASCII character. The integer must be between 0 and 255. The utility assigns ascending collating values, starting from zero, to the literals or integers in the list: the collating value of the first literal or integer is zero, of the second literal or integer is one, and so on. For example, the following are ways you can assign J the collating value 0 and 5 the collating value 1:

```
TABLE NEW_VAL IS "J", "5".
TABLE NEW_VAL IS "J", 53.
TABLE NEW_VAL IS 74, 53.
TABLE NEW_VAL IS 74, "5".
```

Because literal can be more than one ASCII character, you can assign the same collating value to more than one character. For example, to assign A, B, and C the collating value 0 and X,Y, and Z the collating value 1, you could specify

```
TABLE NEW_VAL IS "ABC", "XYZ".
```

Options two and three provide you with alternate ways to represent character lists. Both are convenient to use for long lists. For example, to assign digits 0 through 9 the collating values 0 through 9, you could specify either

```
TABLE NEW_VAL IS 48, 49, 50, 51, 52, 53, 54, 55, 56, 57
```

or

```
TABLE NEW_VAL IS "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
```

You can represent this list more easily by using option two:

TABLE name IS { integer / "literal" } - { integer / "literal" } [, { *integer* / *"literal"* } - { *integer* / *"literal"* }] ...

In option one, literal consists of one or more ASCII character. In option two, however, literal is only one ASCII character. The dash stands for all the characters between and including the two literals or integers. In the example above, you can use either of the following shorthand ways to assign 0 through 9 the collating values 0 through 9:

```
TABLE NEW_VAL IS 48-57.
```

or

```
TABLE NEW_VAL IS "0"-"9".
```

In summary, option two in the TABLE name IS phrase is shorthand for assigning a list of characters a *range* of values. The phrase's option three is another type of shorthand. It assigns a list of characters a *single* value. Option three is

TABLE name IS { integer / "literal" } : { integer / "literal" } [, { *integer* / *"literal"* } : { *integer* / *"literal"* }] ...

The literal in option three, like the literal in option two, can be only one ASCII character. The : stands for all the characters in between the two integers or the two literals.

Here's an example of the long way to assign A through Z the collating value 0:

```
TABLE NEW_VAL IS "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
```

Use option three to specify the shorthand way:

```
TABLE NEW_VAL is "A":"Z".
```

or

```
TABLE NEW_VAL IS 65:90.
```

You can combine all three forms of the TABLE name IS clause in one declaration. For example, if you want to assign

the character(s)	**the collating value(s)**
$	0
0 through 9	1 through 10
uppercase ASCII letters	11

you could specify:

```
TABLE NEW_VAL IS "$", "0"-"9", "A":"Z".
```

To see more clearly why the letters A through Z gets the collating value 11, we'll write this example out the long way:

```
TABLE NEW_VAL IS "$", "0", "1", "2", "3", "4", "5", "6", "7", "8",
      "9", "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
```

Note that "ABCDEFGHIJKLMNOPQRSTUVWXYZ" is the 12th item in the list. Because the first item receives the collating value 0, the 12th item gets the collating value 11.

In this example, what collating values do a through z, and special characters besides $ receive? First, let's answer the question generally. We call these characters the *unmentioned* characters, since you did not include them in the list following the TABLE name IS clause. In the ASCII character code set, each character has a fixed position based on its decimal equivalent. When you explicitly change the collating values of some of the characters, you implicitly change the collating values of the unmentioned characters. But you do not change their relative positions. The unmentioned characters' collating values start from one greater than the highest collating value of the characters in the list. To repeat, their relative positions do not change. Now we can answer the question specifically.

Again, what collating values do a through z, and special characters besides $ receive? Relative to the rest of the characters in the ASCII character set, null is the first. Thus, null is the first in the series of unmentioned characters. Because the greatest collating value of the characters in the list is 11, and the unmentioned series starts from one greater than this number, null receives the collating value 12. The next character in the unmentioned series, CTRL-A, receives the collating value 13. This assignment of ascending values continues to # , which receives the collating value 47. We skip $, a character already explicitly assigned the collating value 0, and continue counting from % to /. Likewise, we skip 0 through 9 and continue counting from : to @ , and skip the uppercase letters and continue counting from [to the final character, DEL.

Here we summarize the values that the letters a through z, and the special characters besides $ receive in this example.

This character	**receives the collating value**
null	12
CTRL-A	13
.	
.	
.	
#	47
.	
.	
.	
%	48
.	
.	
.	
/	58
.	
.	
.	
: (colon)	59
.	
.	
.	
@	65
.	
.	
.	
[	66
.	
.	
.	
a	72
.	
.	
.	
z	97
.	
.	
.	
DEL	102

In general, the collating values of unmentioned characters depend on their relative positions in the ASCII character set.

UNMENTIONED

The answer to the question about what happens to the unmentioned characters brings us to format one's UNMENTIONED. You use UNMENTIONED to assign one collating value to all the characters not mentioned in your declaration. The value assigned depends on the collating values of the mentioned characters and the position of UNMENTIONED in the declaration. Let's look at some examples to see how it works.

Say you are concerned only with sorting records beginning with numeric fields. You want to assign 0 through 9 the collating values 0 through 9, respectively, and assign all other ASCII characters the collating value 10. With these collating values, the utility will output all records beginning with numeric fields, in sorted order, before all other records, which will be unsorted. The following TABLE declaration accomplishes this collating:

```
TABLE NEW_VAL IS "0"-"9", UNMENTIONED.
```

Changing the declaration to

```
TABLE NEW_VAL IS "0"-"8", UNMENTIONED.
```

assigns unmentioned characters the collating value 9.

Changing the position of UNMENTIONED changes the collating value of unmentioned characters. For example, say you want to asssign 0 through 9 the collating values 1 through 10, respectively, and all other ASCII characters the collating value 0. You could specify:

```
TABLE NEW_VAL IS UNMENTIONED, "0"-"9".
```

Note that UNMENTIONED is especially useful in TABLE declarations you plan to use with the COMPRESS massage statement (described in Chapter 6). COMPRESS condenses a record by deleting all characters with collating value zero.

You are not restricted to placing UNMENTIONED at the beginning or the end of the character list; you can place it anywhere in the list. For example:

```
TABLE NEW_VAL IS "A"-"Z", UNMENTIONED, "0":"9".
```

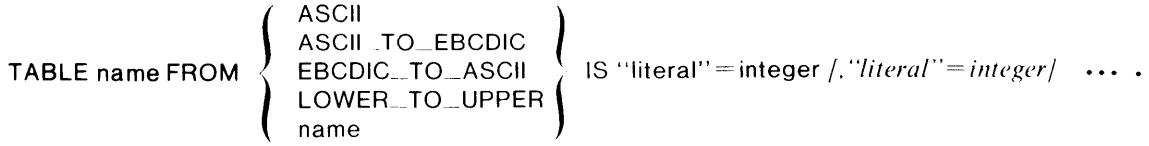
In this case, you assign the uppercase ASCII letters the collating values 0 through 25 and all the unmentioned characters the collating value 26. You assign 0 through 9 the collating value 27.

You can specify each ASCII character and UNMENTIONED only once. Therefore, while you can assign a single collating value to more than one character, you cannot assign more than one value to any single character.

Format Two

For easy reference, here's format two again:

FORMAT TWO



As in option one of format one, literal can be more than one ASCII character. Argument integer is the decimal equivalent of an ASCII character.

Format two works differently from format one. In the first, you *implicitly* alter collating values of unmentioned characters by *explicitly* altering collating values of the characters in the list. In the second format, explicitly altering collating values of mentioned characters does not alter any other character's collating value.

The utility supplies four collating value tables:

- ASCII, which assigns each ASCII character its standard collating value
- ASCII_TO_EBCDIC, which assigns each ASCII character the collating value of its corresponding EBCDIC character
- EBCDIC_TO_ASCII, which assigns each EBCDIC character the collating value of its corresponding ASCII character
- LOWER_TO_UPPER, which assigns each lowercase ASCII character the collating value of its corresonding uppercase letter

Argument name lets you supply your own collating value table.

The IS clause lets you assign to individual characters single collating values, without changing the other collating values in one of the predefined tables or a table that you supply. The utility assigns the collating value you select as integer to the character you select as literal. For example:

```
TABLE NEW_VAL FROM ASCII IS "A"=48, "B"=49, "C"=50.
```

In this case, Sort/Merge assigns A the collating value 48, B the collating value 49, and C the collating value 50. Explicitly changing the collating values of A, B, and C does not change the collating values of 0, 1, and 2. Further, all the other ASCII character's collating values remain the same.

You can let literal be more than one ASCII character. For example, if you want to assign A, B, C, and 0 the collating value 48, and X, Y, Z, and 1 the collating value 49, then you could use this TABLE declaration:

```
TABLE NEW_VAL FROM ASCII IS "ABC"=48, "XYZ"=49.
```

Instead of using one of the utility's predefined tables, you can supply your own. You defined this table in an earlier TABLE declaration. For example, if NEW_VAL defined in the last example is the table you supply, you might want to adjust it slightly. Let's say you want to assign D, in addition to A, B, C, and 0, the collating value 48. You could specify

```
TABLE ADJUST FROM NEW_VAL IS "D"=48.
```

Format Three

We repeat the format here for your convenience:

```
TABLE name1 IS FILE "name2".
```

This format creates a table, name1, from the contents of an AOS file whose pathname is name2. The utility creates the table directly from the first 256 bytes of the file as follows: null translates to the value of the first byte of the file, CNTL-A translates to the value of the second byte, and so on.

We provide format three principally for those who use Sort/Merge as a building block in a larger program system. We caution you: this format can lead to command files that are difficult to understand and maintain.

KEY Declaration

The KEY declaration defines a key field on which the utility bases the sort or merge. You can define more than one key field by using more than one KEY declaration. If you do, the utility treats the first KEY declaration as the primary key, the second as the secondary key, the third as the tertiary key, and so on. You saw two examples of multiple keys in Chapter two: the birthday sort (third example) and the male and female student sort (last example).

Those examples showed how multiple KEY declarations can resolve key conflicts. Here is a similar example. You might want to sort a file containing names by last name first and first name second. Three records in the file could be

```
Prolman     Michael
Prolman     Gerald
Prolman     Maxine
```

A KEY declaration for the last name field allows Sort/Merge to distinguish between the PROLMAN last name and any other last name, but not to distinguish between Michael, Gerald, and Maxine. A second KEY declaration for the first name field resolves the conflict. The utility sorts the records based on the primary (last name) key first, and the secondary (first name) key second.

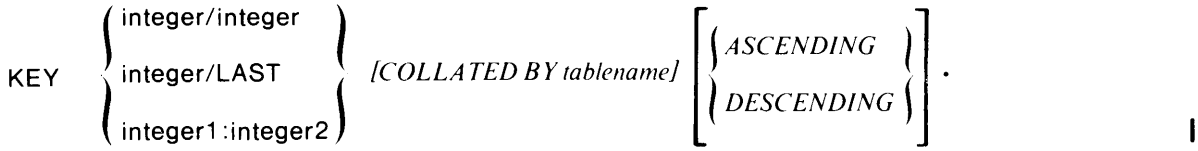
After the utility finishes sorting, you'll find the records ordered

```
Prolman     Gerald
Prolman     Maxine
Prolman     Michael
```

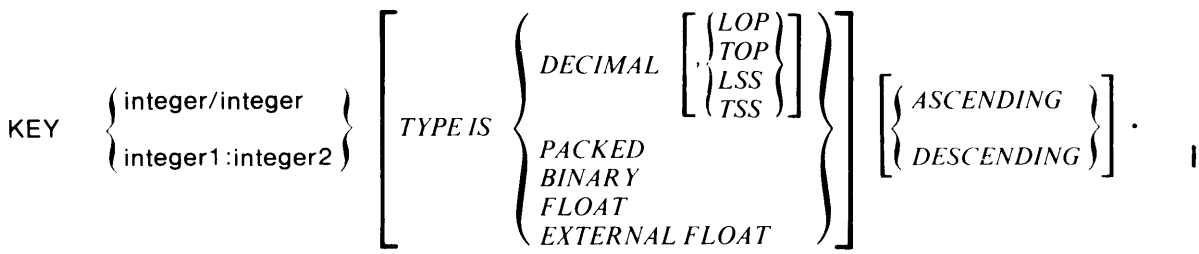
There are two KEY declaration formats. You use format one to define keys containing character data which you want sorted in alphabetical order. You use format two to define keys containing numerical data which you want sorted in numerical order.

The two formats for the KEY declaration are

FORMAT ONE



FORMAT TWO



By default, the utility sorts and merges records in ascending order, using the entire record as the key. The location phrase in both formats restricts a key to a particular character or range of characters, and thereby defines a key field. In the Prolman record example above, the last name field is characters 1 through 12, and the first name field is characters 14 through 25. The corresponding location phrases are 1/12 and 14/25. The KEY declarations which make the last name field the primary key and the first name field the secondary key are

```
KEY 1/12.
KEY 14/25.
```

Format One

COLLATED BY Phrase

The COLLATED BY phrase supplies the utility with a table of collating values defined in a TABLE declaration. When Sort/Merge compares key fields defined in the KEY declaration, it uses the collating values defined in tablename. The utility does not permamently translate character values to those defined in the TABLE declaration. Instead, it temporarily translates character values while comparing key fields. When the utility finishes comparing the key fields, the character values return to whatever they were originally.

The tablename can be one of the predefined tables:

- ASCII_TO_EBCDIC
- EBCDIC_TO_ASCII
- LOWER_TO_UPPER

For example:

```
KEY 1/LAST COLLATED BY LOWER_TO_UPPER.
```

This KEY declaration causes the utility to compare lowercase and mixed uppercase and lowercase text as if they were all uppercase. For instance, CAT, Cat, and cat would all compare equally.

Location Phrase

In addition to the integer/integer and integer/LAST forms of the location phrase, the KEY declaration also uses an integer1:integer2 form. Argument integer1 is the starting byte (or character) of the key field. Argument integer2 is the *field length* (not the ending byte or character). There is no advantage to using this form over the other two, in format one. However, there is an advantage to using this form in format two.

Here are some examples of correspondences between the integer/integer and integer1:integer2 forms:

integer/integer	integer1:integer2
20/30	20:11
22/25	22:4
20/20	20:1

There is no correspondence between the integer/LAST and integer1:integer2 forms.

Format Two

TYPE IS Clause

The default key type is *character* . The TYPE IS clause lets you choose another key type. You can choose between unsigned and signed numeric characters for the key type. The format word DECIMAL by itself indicates an unsigned numeric character field. For example:

```
KEY 1/25 TYPE IS DECIMAL.
```

To indicate a signed numeric character field, use DECIMAL with one of the following:

- LOP (Lead Overpunch)
- TOP (Trailing Overpunch)
- LSS (Lead Separate Sign)
- TSS (Trailing Separate Sign)

For example:

```
KEY 1/25 TYPE IS DECIMAL, TSS.
```

Your other choices for key types are

- PACKED (Packed Decimal Format)
- BINARY
- FLOAT (Internal Floating Point)
- EXTERNAL FLOAT (External Floating Point)

You must use either the integer/integer or the integer1:integer2 form of the location phrase with the TYPE IS phrase.

For more information on the various representations of numbers, see Chapter five of the *COBOL Reference Manual (AOS)*

Location Phrase

The integer1:integer2 form of the location phrase works exactly as it does in format one: integer1 is the starting byte; integer2 is the field length. However, in format two, integer2 does not represent a constant unit of length across all data types. For example, the BINARY data type's unit of length is 1 byte. But the PACKED data type's unit of length is 1 digit.

The data types and their corresponding units of length are

Data Type	Unit	Unit Length
BINARY	byte	1 byte
FLOAT	byte	1 byte
EXTERNAL FLOAT	character	1 byte
DECIMAL	digits	1 digit = 1 byte

Data Type	Unit	Unit Length
DECIMAL, LOP	digits	1 digit = 1 byte
DECIMAL, TOP	digits	1 digit = 1 byte
DECIMAL, LSS	digits	1 byte for sign + 1 byte per digit
DECIMAL, TSS	digits	1 byte for sign + 1 byte per digit
PACKED	digits	1/2 byte for sign + 1/2 byte per digit + if needed, extra 1/2 byte to pad to full byte boundary

The advantage of using the integer1:integer2 form is that you can specify a PACKED data type for a packed field with an even number of digits.

Table 4-2 shows some examples of the correspondence between the integer/integer and the integer1:integer2 forms of the location phrase for each data type other than CHARACTER.

Table 4-2. Correspondences between Location Phrases

Data Type	integer/integer	integer1:integer2
BINARY FLOAT EXTERNAL FLOAT	1/10	1:10
	15/20	15:6
DECIMAL (LOP, TOP)	23/30	23:8
DECIMAL (TSS, LSS)	1/10	1:9
	15/20	15:5
	23/30	23:7
PACKED	5/9	5:9
	Not possible	6:4
	10/11	10:3
	Not possible	1:0

Changing the Collating Sequence

There are three ways to change a collating sequence:

1. change collating values

2. explicitly tell the utility to order records in an ascending or descending sequence

3. do 1. and 2.

Use the COLLATED BY phrase to change collating values. To order records in a descending sequence, use DESCENDING in the KEY declaration. For example:

```
KEY 50/LAST COLLATED BY TABLE_1 DESCENDING.
```

The utility orders records in ascending sequence by default. You can explicity tell the utility to do this by using ASCENDING in the KEY declaration. For example:

```
KEY 50/LAST COLLATED BY TABLE_1 ASCENDING.
```

WORK FILE Declaration

Sort/Merge uses work files for intermediate scratch storage. If you do not declare a work file, the utility builds two default work files in your current working directory and deletes them when processing is done. The WORK FILE declaration lets you define a work file that either the utility creates or you create.

Why would you want to define your own work files? AOS by default creates work files with a small element size. However, AOS works most efficiently with files of a large element size. Thus, it's to your advantage to create work files with a large element size. To control the element size, create the work file(s) with the CLI CREATE command before you invoke Sort/Merge. The *Command Line Interpreter User's Manual* describes the CREATE command; Appendix E further discusses the advantages of creating your own work files.

The format of a WORK FILE declaration is

```
WORK FILE IS "filename".
```

The filename argument must be an AOS pathname. For example:

```
WORK FILE IS ":UDD:MONSTER:WINGED:RODAN".
```

or equivalently,

```
WORK FILE IS "RODAN".
```

(The pathname need not be complete.)

If you declare a work file and that work file does not exist, then Sort/Merge creates it. If you declare more than one work file, the utility will try to use them alternately in a "round-robin" fashion. The utility will not delete any work files named in WORK FILE declarations.

End of Chapter

Chapter 5

Imperatives

An imperative tells the utility how to order input records for the output file. Each command file must contain exactly one imperative.

This chapter explains how each imperative orders input records and some of the reasons for using one over another.

SORT and TAG SORT Imperatives

Both the SORT and TAG SORT imperatives direct the utility to collate input records based on one or more key fields. Remember, if you don't define a key field, the utility uses the entire record as the default key. Also, Sort/Merge collates the records in ascending order, unless you state otherwise in a KEY declaration.

These two imperatives differ in how they use the input records in the sorting process. SORT directs the utility to carry the entire input record through all its process phases. TAG SORT directs the utility to carry a 6-character binary tag and the key fields, instead of the entire record, through all its process phases (except output). Sort/Merge then outputs the entire record as it does for a SORT.

There are two more points to note about TAG SORT. First, the input file for a TAG SORT cannot be a tape or INFOS II file. Second, input file massage statements generally have no impact on the output file if TAG SORT is the imperative. (Chapter 6 describes massage statements.) However, there are two cases when you'll see changes made to input file records by input file massage statements:

- you massage a key field on which the utility bases the TAG SORT. In that case, the massage affects the order of output records, but not their contents.

- you use the IF massage statement to send the massaged records to a *skip file* (separate file). In this case, the records sent to the skip file show the effects of massaging, while those sent to the main output file do not.

How do you decide whether to use SORT or TAG SORT? For most sort applications, you'll want to use SORT instead of TAG SORT. You might consider TAG SORT if one or both of the following conditions are true:

- the length of each record exceeds 512 characters

- available disk space is less than two times the size of the input file (space needed for the work files) plus the output file. (Of course you don't need any disk space for a tape output file.)

We restate this condition as a formula:

Available Disk Space $<$ 2 (Input File Size) + Output File Size

This formula is only a rough rule of thumb.

STABLE SORT and STABLE TAG SORT Imperatives

The STABLE SORT and STABLE TAG SORT imperatives work like SORT and TAG SORT with one exception. When the utility encounters one or more sort keys exactly equal in value, it writes the records containing the duplicate keys to the output file in the same order in which it encountered the duplicate keys. Contrast this with how the SORT and TAG SORT work: when the utility encounters duplicate keys, it may or may not write the records containing the duplicate keys in the same order in which it encounters them.

The same two points which apply to TAG SORT (described in the previous section) also apply to STABLE TAG SORT. First, the input file for a TAG SORT cannot be a tape or INFOS II file. Second, input file massage statements have no impact on the output file unless

- you massage a key field
- you use the IF massage statement to send massaged records to a skip file

STABLE SORT and STABLE TAG SORT usually are slower than SORT and TAG SORT.

MERGE Imperative

The MERGE imperative combines a minimum of two input files into a single output file. The utility bases the merge process on the key field(s) that you define in your command file. You must already have sorted the input files by the same key field(s) before you merge them. See "New Student Merge" in Chapter 2 for an example.

DELETING DUPLICATES Imperatives

The DELETING DUPLICATES imperative directs the utility first to look at all the records' keys. Then the utility discards all but one of a set of records with identical keys.

The format for the DELETING DUPLICATES imperative is

```
{ SORT       }
{ TAG SORT   }
{ STABLE SORT     } [ DELETING DUPLICATES ] .
{ STABLE TAG SORT }
{ MERGE      }
```

Let's take an example. Suppose two census takers visit the same family. As a result, part of the unsorted master file at the census bureau might look like this:

BOURKE	ROBERT	45	NEUROSURGEON	33 MAIN
BOURKE	MARLENE	37	STATIONER	33 MAIN
BOURKE	JARON	13	STUDENT	33 MAIN
BOURKE	ANDREW	11	STUDENT	33 MAIN
BOURKE	JARON	13	STUDENT	33 MAIN
BOURKE	ANDREW	11	STUDENT	33 MAIN
BOURKE	MARLENE	37	HOUSEWIFE	33 MAIN
BOURKE	ROBERT	45	DOCTOR	33 MAIN

Duplicate records need not be identical character for character. As long as the key fields of two records are identical, the two records are identical from Sort/Merge's point of view. The utility groups families together in the sorted master file if the last name field is the primary key, the address field is the secondary key, and the first name field is the tertiary key. (This assumes that a family lives together at the same address.) Because the two sets of Bourke records differ only in the profession field (and not the key fields), there are four duplicate Bourke records.

Fortunately, the bureau knows that sometimes census takers duplicate their routes. And they know that a different person might be home each time a different census taker comes to the same address. Therefore, they want to delete all possible duplicate records from the master file.

To delete all the duplicate records, use one of the DELETING DUPLICATES imperatives for sorts.

If the utility uses SORT DELETING DUPLICATES or TAG SORT DELETING DUPLICATES, then the Bourke names might appear in the output file as follows:

BOURKE	ANDREW	11	STUDENT	33 MAIN
BOURKE	JARON	13	STUDENT	33 MAIN
BOURKE	MARLENE	37	HOUSEWIFE	33 MAIN
BOURKE	ROBERT	45	DOCTOR	33 MAIN

We say "might appear" because the utility does not discard duplicate records in a consistent way when you use these imperatives. The utility might have chosen the records that identify Marlene as a a stationer and Robert as a neurosurgeon.

STABLE SORT DELETING DUPLICATES and STABLE TAG SORT DELETING DUPLICATES give the utility a consistent way to discard duplicate records. The utility retains the first duplicate record it encounters. In this case, the Bourke family will appear in the output file as follows:

BOURKE	ANDREW	11	STUDENT	33 MAIN
BOURKE	JARON	13	STUDENT	33 MAIN
BOURKE	MARLENE	37	STATIONER	33 MAIN
BOURKE	ROBERT	45	NEUROSURGEON	33 MAIN

COPY Imperative

The COPY imperative writes the records in one or more input files to a single output file, and it does this in the order in which Sort/Merge encounters them. If you're copying AOS input files into an AOS output file, you need not specify a KEY declaration, since the utility neither sorts nor merges the records. When used this way, the COPY imperative conveniently lets you massage records without sorting or merging them.

You can also copy AOS files into an INFOS II file, INFOS II files into an INFOS II file, and an INFOS II file into an AOS file. Chapter 8 describes in detail how to do this. Table 5-1 briefly tells you what declarations you'll need for each copy process.

Table 5-1. Declarations for Copy Processes

Copy Process	Declarations
AOS into AOS	AOS INPUT FILE AOS OUTPUT FILE
AOS into INFOS II	exactly one KEY AOS INPUT FILE OUTPUT INFOS INDEX
INFOS II into INFOS II	exactly one KEY INPUT INFOS INDEX OUTPUT INFOS INDEX or OUTPUT INFOS INVERSION
INFOS II into AOS	INPUT INFOS INDEX AOS OUTPUT FILE

End of Chapter

Chapter 6

Massage Statements

Massage statements edit records in two ways. They either manipulate a record's characters or exclude certain records from a Sort/Merge process. We call the massage statements' actions *massaging*. All massage statements (except INSERT TAG) can edit both input and output records. Massage statements for input records go before the imperative, and those for output records go after the imperative. Input massage statements do not affect records in the input file; throughout a Sort/Merge process the input file remains the same. Instead, the utility massages copies of the input file records.

When you massage input records, be certain that keys defined in KEY declarations are in the correct locations after the massage. In other words, be sure you know what the input records will look like after the input massaging, and check to see if key fields defined in KEY declarations are still correct.

Sort/Merge executes massage statements and the imperative in the order in which they appear in the command file. Also, the utility executes all the input file massage statements for one input file record before it executes them all for the next. It does not execute one input file massage statement for all the records in the input file and then execute the next input file massage statement for all the input file records, and so on. Sort/Merge follows the same procedure for output file massage statements.

REFORMAT Massage Statement

The REFORMAT statement moves, repeats, or deletes specific fields within a record. The REFORMAT statement's format is

```
REFORMAT  {integer/integer}  ,  {integer/integer}  [,...]  .
          {integer/LAST   }     {integer/LAST   }
```

The location phrase identifies the field you want to massage. LAST represents a record's last character; thus, you must use the integer/LAST form for a file of records whose last-character positions vary because the records' lengths vary.

The examples illustrating the REFORMAT statement massage this record:

```
1    5    9              20   24       31   35                        55     60
|    |    |              |    |        |    |                         |      |
1A357□□□CLOVER□HONEY□□□FOODMART□□□374□SOUTH□ST.□NEEDHAM□□□□□
```

To move a field, simply rearrange its position in the REFORMAT statement. For example, to move the field 1A357 to the end of the record, you specify

```
REFORMAT 6/LAST, 1/5.
```

or

```
REFORMAT 6/60, 1/5.
```

To move the field CLOVER HONEY to the end of the record, you could specify

```
REFORMAT 1/8, 21/LAST, 9/20.
```

To delete a field, exclude it from the REFORMAT statement. For example, to delete the field 1A357, you could specify

```
REFORMAT 6/LAST.
```

To delete the field CLOVER HONEY, you could specify

```
REFORMAT 1/8, 21/LAST.
```

To repeat a field, list it more than once in the REFORMAT statment. For example, to repeat the field 1A357, you could specify

```
REFORMAT 1/5, 1/5, 6/LAST.
```

Note that you normally would not want to move a field after the delimiter (AOS default or user-defined) of a data-sensitive record. For example, if the last character of the record were NEW LINE, you wouldn't move field 1A357 after NEW LINE. To move a field after a delimiter, you should

1. replace the delimiter with a nondelimiter character by using a REPLACE statement (described in this chapter)

2. reformat the record as you had intended by using a REFORMAT statement

If you wish, you can reinsert any delimiter by using the INSERT statement (described in this chapter).

TRANSLATE Massage Statement

The TRANSLATE statement converts records from one character set to another, or from lowercase ASCII characters to their uppercase equivalents. The character set may be ASCII, EBCDIC, or any other collating sequence you define in a TABLE declaration.

The format of the TRANSLATE statement is

```
TRANSLATE { integer/integer } USING { ASCII_TO_EBCDIC  }
          { integer/LAST    }       { EBCDIC_TO_ASCII  }
                                    { LOWER_TO_UPPER   }
                                    { tablename        } .
```

The location phrase determines which characters of each record that the utility translates. The integer/LAST location phrase is particularly useful when you're translating variable-length records.

The utility supplies three predefined translation tables:

- ASCII_TO_EBCDIC, which translates each ASCII character to its corresponding EBCDIC character
- EBCDIC_TO_ASCII, which translates each EBCDIC character to its corresponding ASCII character
- LOWER_TO_UPPER, which translates each lowercase ASCII letter to its corresponding uppercase letter

The argument tablename lets you supply your own translation table, which you previously must have defined in a TABLE declaration. All the table names must consist only of uppercase letters, digits, and/or the underline.

The TRANSLATE statement uses the same predefined tables as the COLLATED BY phrase of the KEY declaration. However, Sort/Merge uses the tables differently for each statement. For the KEY declaration, the utility *temporarily* translates character values while comparing key fields. This translation does not physically change characters; the output file records are the same as the input file records. For the TRANSLATE statement, the utility *permanently* translates characters; thus, the output file records will not be the same as the input file records.

Let's use the LOWER_TO_UPPER table in an example. Suppose a zoo has an input file of the lions' behavior records, which in part contains

```
Clarence    Very sedate. Clumsily pursues moving objects due to eye problem.
Elsa        Spends most of time sleeping due to illness.
Limpy       Normal behavior.
de LEO      Tempermental. Attacked, killed, and ate head feeder.
```

The zoo's animal psychologist decides to translate all the lions' names to uppercase. Because the name field is character positions 1 through 8, the translate statement she'll need is

```
TRANSLATE 1/8 USING LOWER_TO_UPPER.
```

After the command file which includes this TRANSLATE statement executes, the output file contains

```
CLARENCE  Very sedate. Clumsily pursues moving objects due to eye problem.
DE LEO    Tempermental. Attacked, killed, and ate head feeder.
ELSA      Spends most of time sleeping due to illness.
LIMPY     Normal behavior.
```

If the zoo's psychologist had used the LOWER_TO_UPPER table in a KEY declaration, then the lions' names would not appear in all uppercase letters. Remember that in a KEY declaration, LOWER_TO_UPPER (a predefined table) makes the utility temporarily translate collating values while it compares key fields.

Note that if you sort the pretranslated file in ascending order, de Leo will be the last record because *d* has a higher collating value than *L* . However, if you sort the translated file in ascending order, DE LEO will be the second record because *D* has a higher collating value than *C* but a lower collating value than *E* .

In most cases, you will want to perform your translation for input file records. For example, if you sort a file of EBCDIC records and intend to output them in ASCII, you'll probably want to place the TRANSLATE statement before the SORT imperative. The utility will sort the records based on ASCII collating values. If you place the TRANSLATE statement after the SORT imperative, the utility first sorts the records based on EBCDIC collating values, then translates them. As a result, the output will be ASCII, but it may not be in the proper ASCII sequence.

If you are not careful, ASCII to EBCDIC or EBCDIC to ASCII translations can lead to other sorting problems. For example, the ASCII NEW LINE character's octal value is 12. When you translate NEW LINE into EBCDIC, its octal value changes to 25. AOS does not recognize 25 octal as a default delimiter.

REPLACE Massage Statement

The REPLACE statement replaces one or more ASCII characters in a record with one or more ASCII characters. For instance, this statement can replace

- one character with another
- two or more characters with a single character
- one character with two or more characters

Its format is

REPLACE [{ *ALL* / *ANY* }] "literal_1" IN { integer/integer / integer/LAST } WITH "literal_2" .

We base most of the examples which illustrate this statement on the following record:

```
     5    10    15    20    25    30    35    40    45    50  55    60
     |    |     |     |     |     |     |     |     |     |   |     |
1A9□28□19B6□□□DEPAF□ENTERPRISES□INC.□□□□MAIN□ST.□U.S.A.□□□□□□
```

If you omit ANY and ALL, then Sort/Merge

- scans each record's characters specified in the location phrase from left to right
- replaces the first occurrence literal_1 with literal_2
- stops scanning

For example, suppose that you want to replace the blank spaces in the field 1A9□28□19B6 with / (slash). You could use these two REPLACE statements:

```
REPLACE "<040>" IN 1/11 WITH "/".
REPLACE "<040>" IN 1/11 WITH "/".
```

The first REPLACE statement replaces the first space in the record with a / and the second replaces the second space with a /.

After the massage, the record is

1A9/28/19B6□□□DEPAF□ENTERPRISES□INC.□□□MAIN□ST.□U.S.A.□□□□□□

In the preceding example, you replaced a single character with another single character. In the next example, you replace a single character with more than one character. If you want to replace each blank space in the field 1A9□28□19B6 with ***, then you could specify

```
REPLACE "<040>" IN 1/11 WITH "***".
REPLACE "<040>" IN 1/11 WITH "***".
```

After the massage, the record is

1A9***28***19B6□□□DEPAF□ENTERPRISES□INC.□□□MAIN□ST.□U.S.A.□□□□□□

The examples shown so far replace one character with one other character or replace one character with many characters. The next example replaces a number of characters with a fewer number of characters. To replace the field 1A9□28□19B6 with 1A92819B6, which in effect removes the blanks, you could specify

```
REPLACE "1A□28□19B6" IN 1/11 WITH "1A92819B6".
```

After the massage, the record is

1A92819B6□□□DEPAF□ENTERPRISES□INC.□□□MAIN□ST.□U.S.A.□□□□□□

A REPLACE statement that includes the ANY phrase can do the work of many REPLACE statements that do not include the ANY phrase. The REPLACE ANY phrase tells the utility to scan the entire field specified in the location phrase once, from left to right, and to replace each occurrence of literal_1 with literal_2 . For example:

```
REPLACE ANY "<040>" IN 1/11 WITH "/".
```

accomplishes in one statement the task performed by these two statements:

```
REPLACE "<040>" IN 1/11 WITH "/".
REPLACE "<040>" IN 1/11 WITH "/".
```

Likewise,

```
REPLACE ANY "<040>" IN 1/11 WITH "***".
```

accomplishes in one statement the task performed by

```
REPLACE "<040>" IN 1/11 WITH "***".
REPLACE "<040>" IN 1/11 WITH "***".
```

The ALL phrase works differently than the ANY phrase. The ALL phrase tells the utility to scan the field defined in the location phrase from left to right, and replace literal_1 with literal_2. The utility then rescans the field from the beginning, replacing literal_1 with literal_2. The utility continues to rescan the field until there are no more occurrences of literal_1.

Let's look at two examples. Suppose that you have this record:

ABBCABBC

and this REPLACE statement:

```
REPLACE ALL "AB" IN 1/LAST WITH "A".
```

When you execute this statement, the utility scans the entire record from left to right, replacing any occurrence of the literal AB with the literal A. The first scan produces

ABCABC

The utility then rescans the entire record, again replacing any occurrence of the literal AB with the literal A. The second scan produces

ACAC

After the second scan, there are no more occurrences of the literal AB. The utility rescans the record a third time and of course finds no occurrence of the literal AB. So the scanning stops.

The next example's REPLACE statement reduces every occurrence of two or more consecutive blank spaces in the ENTERPRISES INC. record to just one blank space. The statement is

```
REPLACE ALL "<040><040>" IN 1/LAST WITH "<040>".
```

After the massage, the record is

1A9□28□19B6□DEPAF□ENTERPRISES□INC.□MAIN□ST.□U.S.A.□

In contrast, if you choose ANY instead of ALL in the above statement

```
REPLACE ANY "<040><040>" IN 1/LAST WITH "<040>".
```

the resulting massaged record is

1A9□28□19B6□□DEPAF□ENTERPRISES□INC.□□MAIN ST.□U.S.A.□□□

You can use the REPLACE statement to remove a literal without replacing it with another literal. Because literal_2 can be "", the null literal, you can replace literal_1 with null. This in effect removes literal_1. For example, to remove the field INC.□□□ from the ENTERPRISES record, you could specify

```
REPLACE "INC.□□□" IN 33/39 WITH "".
```

After the massage, the record is

1A9□28□19B6□□□□DEPAF□ENTERPRISES□□MAIN□ST.□U.S.A.□□□□□□

REPLACE TABS Massage Statement

The REPLACE TABS statement replaces tab stops in a record with one or more ASCII characters. This statement is especially useful when you want to enter records with many columns from your terminal. Instead of spacing over to each column when you enter records, you use tab stops, and then later replace the tab stops with characters.

It's easy to forget that TAB is only one character. You might therefore have incorrect location phrases in your KEY declarations. For example, you could type the following record with column starting positions of 1, 17, 33, and 49:

```
9A542        desk        2 drawer        brown
```

If you use tab stops to separate the columns, then the 9A542 field begins in character position 1, the desk field begins in character position 7 (not 17), the 2 drawer field begins in character position 12 (not 33), and the brown field begins in character position 21 (not 49). If you use the KEY declaration

```
KEY 49/55.
```

to sort the above record (and others like it) by color, you'll be wrong. Not only does the color field begin in another character position (21), but also there is no field starting at character position 49.

To remedy the problem, simply use REPLACE TABS to substitute characters for the TABs. The space character is often appropriate. The utility automatically inserts the correct number of characters to align the columns in the desired character positions.

The format of the REPLACE TABS statement is

REPLACE TABS IN { integer/integer / integer/LAST } WITH { "literal" / integer } *[, TAB STOPS ARE integer [, integer] ...]* .

REPLACE TABS IN Phrase

The REPLACE TABS IN phrase replaces tab stops occurring in the range specified by the location phrase with one or more ASCII characters. These replacement characters are either **integer**, the decimal equivalent of an ASCII character, or **literal**. Both **literal** and its decimal equivalent must be only one character. The **integer** must be in the range of 0 through 255, inclusive.

TAB STOPS ARE Clause

The TAB STOPS ARE clause lets you define your own tab stops. You cannot define more than twenty; the utility ignores TABs occuring beyond the last defined tab stop. If you don't use this clause, Sort/Merge uses default tab stops at columns 9, 17, 25, 33, ..., and 129.

Examples

All the examples massage this record:

THE < 011 > COW JUMPED < 011 > OVER THE < 011 > MOON.

< 011 > is the octal equivalent of the TAB character.

If you want to replace all TABs with dashes (-), then you could use the following statement:

```
REPLACE TABS IN 1/LAST WITH "-".
```

After the massage, the record is

```
1        9                  25              41
|        |                  |               |
THE-----COW JUMPED------ OVER THE--------MOON.
```

The utility automatically aligned the fields on the default tab stops: 9, 25, and 41.

Suppose again that you want to replace all the TABs with a character. But this time you want to fill the tab stops with asterisks and define your own tab stops. You could specify:

```
REPLACE TABS IN 1/LAST WITH 42, TAB STOPS ARE 12, 24.
```

(The asterisk's ASCII decimal equivalent is 42.)

After the massage, the record is

```
1           12             24
|           |              |
THE********COW JUMPED**OVER THE<011>MOON.
```

Note that the massage did not replace the last TAB. That's because the TAB between "THE" and "MOON" is in a character position beyond the tab stops you defined.

Let's say that you want to replace all TABs located only in character positions 10 through 25 with ampersands (&). You also want to define tab stops at character positions 15 and 20. You could use the following statement:

```
REPLACE TABS IN 10/25 WITH "&", TAB STOPS ARE 15, 20.
```

After the massage, the record is

```
                    20
                    |
THE<011>COW JUMPED&&&&&OVER THE<011>MOON.
```

Note that "OVER" starts on character position 20, a defined tab stop, and not 25, a default tab stop.

PAD Massage Statement

The PAD statement converts a variable-length or data-sensitive record to a record of fixed length by padding it with any character you select. Don't confuse a record of fixed length with a fixed-length record. Variable-length records retain their type's characteristics but become all the same length; data-sensitive records retain their type's characteristics but become all the same length.

PAD's format is

```
PAD TO integer CHARACTERS WITH { "literal" }
                               { integer   }
```

The length that you specify as integer in integer CHARACTERS must be *at least* as long as the longest record in the input file. The padding character is either literal, an ASCII character or its octal equivalent, or integer, the decimal character code of an ASCII character. The literal can be only one character, and the integer must be in the range of 0 and 255, inclusive. For example, suppose that you have the following data-sensitive records which you want to pad with asterisks to character position 75:

```
SHARK LOAN CO.     $400.00   25%   JAY'S FRUIT STAND
QUICK CASH CO.     $750.00   30%   TOWN BOWLING
CASH-IN-A-PINCH    $250.00   35%   LIQUER AND WINES IMPORT
```

There are three ways to construct the PAD statement. Each way depends on which argument you choose in

```
WITH { "literal" }
     { integer   }
```

If you let literal equal *, then the PAD statement is

```
PAD TO 75 CHARACTERS WITH "*".
```

If you let literal equal < 052 > , the octal value of the asterisk, then the PAD statement is

```
PAD TO 75 CHARACTERS WITH "<052>".
```

If you let integer equal 42, the character code of the asterisk character, then the PAD statement is

```
PAD TO 75 CHARACTERS WITH 42.
```

The utility begins padding after the last character in the record. Because the last character of a data-sensitive record is the delimiter, the utility will pad the first record after the delimiter. But then the record will no longer be data-sensitive since the delimiter will not be the last character of the record. Thus, to pad data-sensitive records and keep them data-sensitive, do the following for each record:

1. replace the delimiter using the REPLACE statement
2. pad the record using the PAD statement
3. insert a "new" delimiter after the last character using the INSERT statement

For our example, we could use these massage statements to pad the data-sensitive records:

```
REPLACE "<012>" IN 1/LAST WITH "".
PAD TO 75 CHARS WITH "*".
INSERT "<012>" AFTER LAST.
```

After you execute a command file which contains these statements, the output file contains

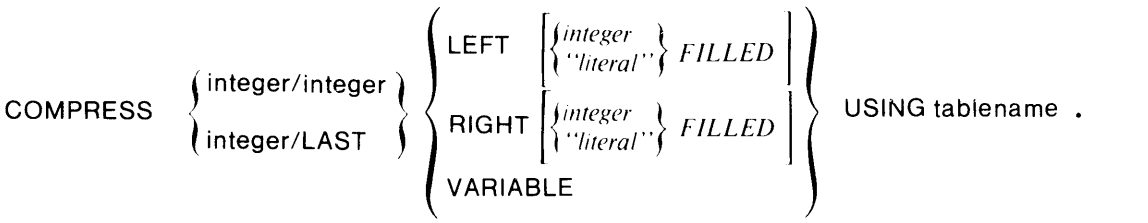
```
SHARK LOAN CO.     $400.00   25%   JAY'S FRUIT STAND ************
QUICK CASH CO.     $750.00   30%   TOWN BOWLING **************
CASH-IN-A-PINCH    $250.00   35%   LIQUOR AND WINES IMPORT ***
```

Note that padding variable-length records to a fixed length does not alter the record format characteristic of the output file. The utility automatically creates the output file with the correct AOS record characteristic, unless you use the appropriate RECORDS ARE clause. In this example, the utility creates the output file with variable-length records.

COMPRESS Massage Statement

The COMPRESS statement removes characters whose collating value is zero. You assign characters the collating value zero in a TABLE declaration (described in Chapter 4).

The format of the COMPRESS statement is



The location phrase indicates the field (range of characters) you want to compress. You can compress the entire record by specifying 1/LAST.

The tablename is the name of the table defined in a TABLE declaration. For example, to assign apostrophe (') the collating value zero, you could specify

```
TABLE IRISH FROM ASCII IS " ' " = 0.
```

The LEFT and RIGHT options justify compressed fields to the left or right, respectively, and fill excess character positions with nulls (0 octal). For example, say that you want to compress all names with an apostrophe, such as O'Leary, and left justify the name field. Removing the ' is desirable because otherwise, all the names starting with O' would appear together instead of in a reasonable alphabetical order. (For example: O'Brien, O'Leary, Ofner instead of O'Brien, Ofner, O'Leary.) You first need to assign ' the collating value zero so that a COMPRESS statement can eliminate ' from the name field. Table IRISH does this. Assume that part of one record is

```
11                                    21
|                                     |
O'LEARY<040><040><040><040>
```

```
COMPRESS 11/21 LEFT USING IRISH.
```

then the record becomes

```
11                                          21
|                                           |
OLEARY<040><040><040><040><000>
```

If you want to right justify the name field, specify

```
COMPRESS 11/21 RIGHT USING IRISH.
```

Then the record becomes

```
 11                                         21
 |                                          |
<000>OLEARY<040><040><040><040>
```

In sum, LEFT pads compressed fields with nulls at the end of the field to left justify. RIGHT pads compressed fields with nulls at the beginning of the field to right justify.

NOTE: Null insertion in data-sensitive records is dangerous because null is one of the AOS default delimiters for data-sensitive records.

If you want to avoid the problem of null insertion in a data-sensitive record, or if you simply want a pad character other than null, then use the form

$$\left[\left[\left\{\begin{matrix}integer\\"literal"\end{matrix}\right\}\right]\ FILLED\right]$$

Select literal , a single character, or integer , its decimal equivalent, as the pad character.

In our O'LEARY record example, if you specify

```
COMPRESS 11/21 LEFT "*" FILLED USING IRISH.
```

then the resulting record is

```
11                                    21
|                                      |
OLEARY<040><040><040><040>•
```

If you don't want the field padded after the compression, use VARIABLE. This option left justifies and truncates (rather than pads) the field. Thus, the record's next noncompressed field begins in the character position immediately following the last character of the compressed field. For example, if you specify:

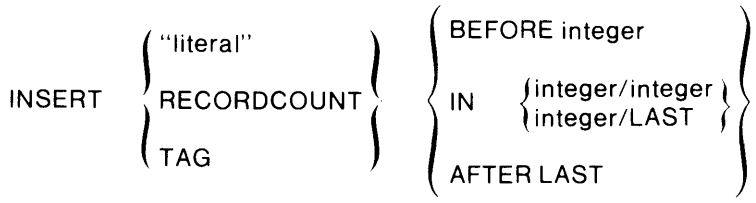
```
COMPRESS 11/21 VARIABLE USING IRISH.
```

then the O'LEARY field becomes

```
11                                  20
|                                    |
OLEARY<040><040><040><040>
```

INSERT Massage Statement

The INSERT statement adds information anywhere in a record. Its format is



Literal Phrase

The literal phrase inserts the characters you choose as literal at the location you indicate. Use the IN, BEFORE integer, or AFTER LAST phrase to indicate where you want to insert the characters. If you use the IN phrase, then the utility

- deletes the characters in the range specified in the location phrase
- inserts literal in the range formerly occupied by the deleted characters
- expands or contracts the record, if necessary, to fit literal

For example, say you have this simple record:

```
123456
```

The INSERT statement

```
INSERT "*" IN 3/3.
```

produces the record

```
12*456
```

The INSERT statement

```
INSERT "*" IN 2/5.
```

produces the record

```
1*6
```

The INSERT statement

```
INSERT "***" IN 3/3.
```

produces the record

```
12***456
```

If you use the BEFORE integer or AFTER LAST phrase, then the utility inserts the literal before character position integer or after the last character, respectively. No characters are deleted. For example, if we use the last example's input record, then the INSERT statement

```
INSERT "*" BEFORE 1.
```

produces the record

```
*123456
```

The INSERT statement

```
INSERT "*" AFTER LAST.
```

produces the record

```
123456*
```

RECORDCOUNT Phrase

If you specify RECORDCOUNT, the utility generates an 8-byte ASCII decimal number that represents each input file record's ordinal position. The utility numbers the records sequentially, beginning with one for the first record in the first input file. It gives the first record in the second input file a record number one greater than the last record in the first input file. Sort/Merge attaches these numbers to the output file records.

As with the literal phrase, you can insert the RECORDCOUNT numbers anywhere in the record.

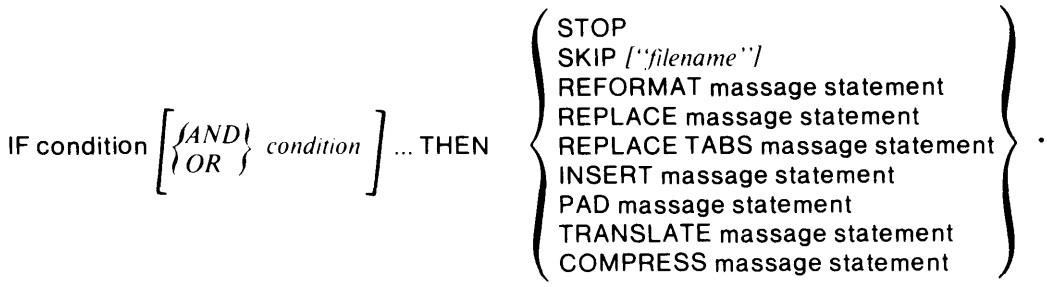
TAG Phrase

The TAG option tells the utility to attach a 6-character binary tag to each record. The tag's first two characters are the number of the file which contains the record; the last four characters are the record's logical disk address. The TAG SORT imperative inserts these tags for you automatically. Therefore, do *not* use INSERT TAG to prepare a record for tag sorting. If you use INSERT TAG, then be sure that the input files are on disk and are not data-sensitive. Unlike all the other massage statements, you may use INSERT TAG only as an input file massage statement.

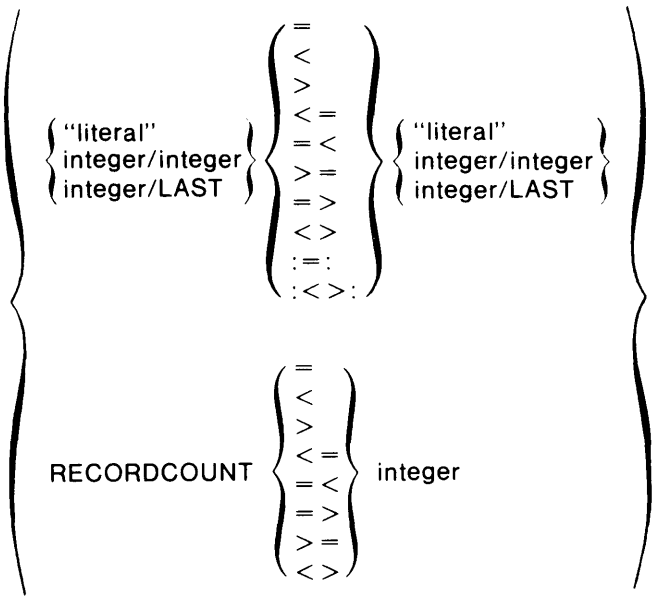
IF Massage Statement

Recall that massage statements either manipulate a record's characters or exclude certain records from a Sort/Merge process. The IF statement establishes conditions which can cause the utility to perform either of these functions. If the condition is true, then the utility performs a specific massage statement, excludes certain records, or stops processing.

The IF statement's format is



The format of the *condition* phrase is:



IF Phrase

The IF phrase, which establishes a condition, is like the IF phrase of many programming languages. If the condition is true, then the utility executes the THEN phrase; if the condition if false, then the utility does not execute the THEN phrase. The IF phrase uses the following operators:

Operator	Means
=	is equal to
<	is less than
>	is greater than
= < or < =	is less than or equal to
= > or > =	is greater than or equal to
<>	is not equal to
:=:	appears in
:<>:	does not appear in

The literal on either side of an operator can be a number.

In general, specify the same length literal or range (defined by the location phrase) on each side of the operator. If the lengths differ, Sort/Merge pads the shorter literal or range to the length of the longer one, using the blank character. The two operators :=: and : < > : allow you to perform *floating compares,* operations which check for the presence or absence of a literal in a range. If you use either of these operators, the utility doesn't pad the length of the shorter literal or range.

Let's look at a few examples. Suppose that you want to perform some process on all records of males in REGISTER_6, the register of sixth grade students we used in the examples of Chapter 2. One of the records from that file is

```
                                        45
                                        |
Prolman     Michael     12/09/70        M       Crocker
```

The record's sex field is character position 45. Thus, you need an IF phrase which establishes the following condition: if the 45th character is M, then execute the THEN phrase. The IF phrase which accomplishes this is

```
IF 45/45 = "M"
```

What if you don't know exactly where in a record a character appears? How do you establish a condition for a character which might or might not appear anywhere in a field? The two operators :=: and : < > : allow you to establish a condition for *floating* characters to do floating comparisons. For example, say that you want to process a record only if the letter A appears in any character position from 10 through LAST. To establish this condition, you specify

```
IF "A" :=: 10/LAST
```

If A appears in any character position from 10 to LAST, then the condition is true and the utility executes the THEN phrase. If you want the utility to process a record only if A does *not* appear in any character postion from 10 through LAST, use this IF phrase:

```
IF "A" :<>: 10/LAST
```

Sometimes your condition won't depend on the contents of a record, but rather on just the number of records the utility processes. For example, you might want to process only the first 500 records of a file. The RECORDCOUNT phrase lets you do this. In this case, the phrase is

```
IF RECORDCOUNT > 500
```

Sometimes you might want to combine conditions. For example, you might want to process a record only if the first character is A *and* the 10th character is +. You can use AND or OR to logically combine more than one condition phrase. As a general rule, AND takes precedence over OR. Also, parentheses tell the utility to test for the truth of whatever appears inside the parentheses before testing for the truth of whatever appears outside them. For example, in the condition

```
IF "A"=5/5 OR "B"=6/6 AND "C"=7/7
```

the utility tests the AND condition before the OR condition. Thus implicitly, this last condition is the same as

```
IF "A"=5/5 OR ("B"=6/6 AND "C"=7/7)
```

If you want the OR condition to take precedence over the AND condition, you must place the parentheses as follows:

```
IF ("A"=5/5 OR "B"=6/6) AND "C"=7/7
```

These two different conditions can lead to two different truth values. For example, if the seventh character of a record does not equal C, then

- in the first case, though ("B"=6/6 AND "C"=7/7) must be false, the whole condition may still be true
- in the second case, because "C"=7/7 is false, the whole condition must be false

THEN Phrase

If the condition that the IF phrase establishes is true, the utility executes the THEN phrase. Note that you cannot use an IF phrase as a THEN phrase; that is, you cannot nest IF statements. The THEN phrase has several options:

- STOP
- SKIP [*filename*]
- Massage Statements

The STOP option tells the utility to stop processing records. If STOP precedes the imperative, then the utility stops reading input records. If STOP follows the imperative, then the utility stops writing output records.

For example, if your IF statement which appears before the imperative is

```
IF RECORDCOUNT > 500 THEN STOP.
```

then the utility looks at only the first 500 records in the input file.

Instead of truncating the input file to a fixed number of records, you might want a particular record, whose position is unknown in the file, to signal the end of the input file. For example, suppose you want to copy all records that contain 1969 or 1970 birthdays. (Assume that the year subfield is character positions 36 and 37, and that the records are already sorted in ascending order by birthday.) To do this, you need the following IF statement:

```
IF 36/37 >= "71" THEN STOP.
```

The utility looks at the first record with a 71 year field, but does not include it in the output file. In addition, the utility will not include any subsequent record in the output file.

The SKIP option tells the utility to ignore certain input file records either before it processes the whole file, or before it writes processed records to the output file. The action it takes depends on whether the IF statement comes before or after the imperative. For example, suppose you want to sort a transaction file by department number (character positions 1 through 4). If you don't want to include the transactions for department A101 in the sort process, then you can specify:

```
IF 1/4 = "A101" THEN SKIP.
SORT.
```

Instead of simply ignoring certain records, you can send them to a separate file, called a *skip file*. In the last example, if you wanted to sort all the input records and then send all records beginning with A101 to a skip file named REJECTS, you could specify:

```
SORT.
IF 1/4 = "A101" THEN SKIP "REJECTS".
```

Because the utility sorts all the input file records before it sends A101 records to REJECTS, all the records in REJECTS will be sorted. If you reverse the order of the imperative and the massage statement, so that the IF statement precedes the SORT imperative, the utility will send the A101 records to REJECTS unsorted.

There are some rules to know about using a skip file:

- do not declare a skip file in input or output file declarations
- it is not a substitute for the output file; if you use a skip file, you must also declare an output file
- it must not exist prior to command file execution; the utility creates it
- its name must be an AOS pathname
- you can name the same skip file in more than one IF statement (even though the utility creates only one skip file with that name), or you can name different skip files in different IF statements. (The number of possible skip files depends on the number of available I/O channels)
- it must be a sequential, disk file. The skip file cannot be an INFOS II file or be on a device (for example, tape).

There is one more option which you may choose. Instead of STOP or SKIP, you can choose any massage statement other than IF. For example, say that some records in a transaction file begin with the department number and some do not. Perhaps this is because the departments' data procedures were not coordinated at one time. In some records, for example, the department number might occur at the end of the record. If you receive these differently formatted records at one central location, you probably will want them all to have the same format before you process them.

The transaction file consists of 66 character data-sensitive records. Some of the records are

```
1   4                                                          62  65
|   |                                                          |   |
A101        SHOES       BLACK          10EE                      4
A101        SHOES       BLACK          9D                        7
DRAPES      GREEN       WINDOW         12                     A102
DRAPES      YELLOW      WINDOW         5                      A102
CLOCK       BLUE        ELECTRIC       14                     A103
CLOCK       TAN         WIND           18                     A103
A101        SHOES       BLACK          9E                        6
PAPER       CREAM       8.5 x 11       55                     A104
```

To move all department numbers appearing in the last four character positions (not including the delimiter) to the first four character positions, you could specify:

```
IF 62/65 = "A102" OR 62/65 = "A103" OR 62/65 = "A104"
      THEN REFORMAT 62/65, 1/61, 66/66.
IF 1/4 = "A102" OR 1/4 = "A103" OR 1/4 = "A104"
      THEN INSERT "   " BEFORE 5.
```

The first IF statement checks to see whether a department number appears in character positions 62 through 65. It doesn't check for the A101 department number because A101 always appears in the first four character positions. If the condition is true, then Sort/Merge executes the REFORMAT massage statement. The second IF statement inserts blanks to align all the columns under the A101 record columns.

After you execute the command file that contains these IF statements, all the records have the A101 record format:

A101	SHOES	BLACK	10EE	4
A101	SHOES	BLACK	9E	6
A101	SHOES	BLACK	9D	7
A102	DRAPES	GREEN	WINDOW	12
A102	DRAPES	YELLOW	WINDOW	5
A103	CLOCK	BLUE	ELECTRIC	14
A103	CLOCK	TAN	WIND	18
A104	PAPER	CREAM	8.5 x 11	55

End of Chapter

Chapter 7

Command Lines

The Sort/Merge command line invokes the utility and controls some of its actions. For example, the command line can direct the utility to send statistical output to the line printer or to suppress it altogether.

The command line must have a *command word* and may have *switches* and/or an INTO FROM phrase. For example, in Chapter 2 we used the command line:

```
)SORT/C=LAST_NAME_SORT/O
```

SORT is the command word; /C=LAST_NAME_SORT and /O are switches.

You can use the switches and the INTO FROM phrase in many different combinations, which fall into three main categories. The categories are

- noninteractive mode
- interactive mode
- command line file declarations

Noninteractive Mode

In noninteractive mode, you use a command file which exists before you invoke the utility. The /C=filename switch names the command file in a command line. Also, the functions of the command line and command file are separate.

We used this mode for the examples in Chapter 2. Most of the time you'll use the noninteractive mode.

The format for a noninteractive mode command line is

$$\left\{\begin{matrix}\text{SORT}\\\text{MERGE}\end{matrix}\right\}\ \text{/C=filename}\ \left[\left\{\begin{matrix}\text{/L}\\\text{/L=}\textit{filename}\end{matrix}\right\}\right]\ [\text{/N}]\ [\text{/O}]\ [\text{/S}]$$

/C=filename	directs the utility to use an existing command file, filename.
/L	sends statistical output and any error messages to the current listfile.
/L=filename	sends statistical output and any error messages to *filename* (e.g., an AOS disk file, @ LPT, @ OUTPUT, or @ MTA0). The utility creates an AOS file if it doesn't already exist.
/N	suspends execution of the imperative. The utility still checks the syntax of the command file statements. Thus, you can use /N to speed debugging a command file.
/O	tells the utility that the output file already exists. The utility deletes and then recreates it with the results of the Sort/Merge process.
/S	suppresses the statistical output.

Command Word and Imperative Relationship

In the noninteractive mode, the command word invokes the utility. But it does not act as a command file imperative. The Sort/Merge process depends solely on the command file imperative.

For example, if the command line is

```
)SORT/C=RECORD_MERGE↵
```

and MERGE is the command file imperative, then the utility merges the input files.

Let's take another example. If the command line is

```
)MERGE/C=RECORD_COPY↵
```

and COPY is the command file imperative, then the utility copies the input file(s).

Detecting Syntax Errors

Sort/Merge detects syntax errors in the command file. After detecting an error, the utility prints an error message on the terminal. For example, the command file TEACHER_SORT contains

```
INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
OUPUT FILE IS "TEACHERS".
KEY 55/68.
SORT.
END.
```

If you type the command line

```
)SORT/C=TEACHER_SORT/O↵
```

then the utility detects an error and displays this message on your terminal:

*** SYNTAX ERROR:*
EXECUTION INHIBITED - ONE ERROR WAS DETECTED

By itself, this information is not very helpful. The error message would not readily lead you to find that "OUPUT" is misspelled. Because "OUPUT" is misspelled, there is no OUTPUT FILE declaration, which in turn causes the command file to be incomplete.

To receive more detailed information about an error, use

- the /L=filename switch in the command line, or
- the /L switch in the command line and set the current listfile to the desired file. Note: use @ CONSOLE in place of @ OUTPUT.

If you use one of these switches, Sort/Merge gives you the following additional information in the error message:

- the line in which the error occurs
- a caret (^) under the probable word causing the error

For example, if you type

```
)SORT/C=TEACHER_SORT/O/L=@CONSOLE)
```

then the utility prints the following on your terminal:

```
Command line:  SORT/C=TEACHER_SORT/O/L=@OUTPUT)
1.  INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
2.  OUTPUT FILE IS "TEACHERS".
      ^

  **SYNTAX ERROR: [LINE 2]

3.  KEY 55/68.
4.  SORT.
5.  END.
EXECUTION INHIBITED - ONE ERROR WAS DETECTED
```

Examples

Suppose that you want to execute a command file named REORDER and want to suppress the statistical output. (This time the output file already exists.) To accomplish this, you need the following command line:

```
)SORT/C=REORDER/O/S)
```

In this next example, instead of suppressing statistical output, you want to send it to the line printer. You already created the output file. You need the following command line:

```
)SORT/C=REORDER/L=@LPT/O)
```

Interactive Mode

In interactive mode, you enter the command file from your terminal. Therefore, you don't name a command file in the command line. After you type an interactive mode command line, the utility returns a * prompt. Type your command file statements next to this prompt.

Sort/Merge informs you about command file syntax errors in the interactive mode the same way that it does in the noninteractive mode. See "Detecting Syntax Errors" in the previous section.

The format for an interactive mode command line is

$$\left\{\begin{matrix}\text{SORT}\\ \text{MERGE}\end{matrix}\right\} /\text{C}\ [/T=\textit{filename}]\ \left[\left\{\begin{matrix}/L\\ /L=\textit{filename}\end{matrix}\right\}\right]\ [/N]\ [O]\ [/S]$$

/C	indicates that you intend to enter a command file at your terminal.
/T=filename	directs the utility to save the command file you type in *filename*. If *filename* already exists, then the utility preserves it and appends the new command file statements.

/L sends statistical output and any error messages to the current listfile.

/L=filename sends statistical output and any error messages to *filename* (e.g., an AOS disk file, @ LPT, @ OUTPUT, or @ MTA0). The utility creates an AOS file if it doesn't already exist.

/N suspends execution of the imperative. The utility still checks the syntax of the command file statements. Thus, you can use /N to speed debugging a command file.

If you include /N without /T=filename, the utility will ask if you want to save your input. Type Y (for yes) or N (for no) in response. If you respond Y, then the utility asks for the name of a file in which to save your command file.

/O tells the utility that the output file already exists. The utility deletes and then recreates it using the results of the Sort/Merge process.

If you don't include the /O switch and the output file already exists, Sort/Merge asks if you want to overwrite the existing file. Type Y (for yes) or N (for no) in response.

/S suppresses the statistical output.

Command Word and Imperative Relationship

As in the noninteractive mode, the command file imperative determines which Sort/Merge process the utility performs. The command word only invokes the utility.

HELP Messages

While typing the command file, you can request HELP messages. Displaying HELP messages won't interfere with the Sort/Merge process. Type HELP (in uppercase only) to get a list of HELP message topics. After you select a topic, type HELP followed by a space and the desired topic:

```
)HELP topic
```

You then receive a brief explanation. For example, if you want a HELP message about KEY declarations after typing

```
*OUTPUT FILE IS "REGISTER_6".↵
```

type

```
*HELP KEY_DECLARATION↵
```

Aborting Interactive Input

If you want to immediately escape from interactive input to the utility, type CTRL-D. The utility will then abort interactive input and you'll return to the CLI.

Examples

This example shows a command line which invokes the interactive mode and a command file typed next to the interactive mode prompt. There is no /O switch in the command line because output file MASTER does not yet exist. The command line is

```
)SORT/C↵
```

The command file that you type next to the prompt is

```
*INPUT FILE IS "REGISTER_6", RECORDS ARE 70 CHARACTERS.
*OUTPUT FILE IS "MASTER".
*KEY 1/12.
*SORT.
*END.
```

In this example, you want to save the statements that you type on your terminal in file SAVE. You're debugging a command file and you don't want to execute the imperative. This time the output file already exists. The command line is

```
)SORT/C/T=SAVE/N/O↵
```

Command Line File Declarations

You must let the utility know what the input and output files are by declaring them. In the noninteractive and interactive modes (the first two categories we looked at), you declare input and output files in the command file. In the third category, you can also declare input and output files in the command line by naming them in the INTO FROM phrase. This phrase leads to many options. The two main options are using a command line with a command file, or without a command file. We'll describe each option in detail and then summarize them in Table 7-1.

Command Line without Command File

It's possible to use the Sort/Merge utility without a command file; the command line can be sufficient. If you don't declare the input and output files in a command file, you must declare them in the command line. You do this by naming the files in an INTO FROM phrase.

The following are the command line formats which allow you to use the Sort/Merge utility without a command file:

SORT $\left[\left\{\begin{matrix}/L \\ /L=filename\end{matrix}\right\}\right]$ *[/N] [/O] [/S]* INTO outfile FROM infile ...

MERGE $\left[\left\{\begin{matrix}/L \\ /L=filename\end{matrix}\right\}\right]$ *[/N] [/O] [/S]* INTO outfile FROM $infile_1$ $infile_2$ ...

/L	sends statistical output and any error messages to the current listfile.
/L=filename	sends statistical output and any error messages to *filename* (e.g., an AOS disk file, @ LPT, @ OUTPUT, or @ MTA0). The utility creates an AOS file if it doesn't already exist.
/N	suspends execution of the imperative. The utility still checks the syntax of the command file statements.
/O	tells the utility that the output file already exists. The utility deletes and then recreates it using the results of the Sort/Merge process.
/S	suppresses the statistical output.
outfile	is the output file.
infile	is an input file.

The INTO outfile replaces the command file's OUTPUT FILE declaration; the FROM infile replaces an INPUT FILE declaration. For example, the command line

```
)SORT INTO TEACHERS FROM REGISTER_6)
```

causes the utility to perform the same actions as this command file:

```
INPUT FILE IS "REGISTER_6".
OUTPUT FILE IS "TEACHERS".
KEY 1/LAST.
SORT.
END.
```

Invoking the utility without a command file imposes many limitations:

- You cannot control the range of characters on which the utility bases the sort or merge. Sort/Merge automatically selects 1/LAST as the location phrase.
- You cannot specify the record type of either the input or the output file in the command line. Their record type must be data-sensitive or fixed-length, or else you'll get an error message telling you that the records are not one of these two types.

  Because text editors change a file to, or create a file with, dynamic records, you cannot create or modify an input file with LINEDIT.
- The command words used with the INTO FROM phrase function differently than the noninteractive and interactive mode command words. Noninteractive and interactive mode command words only invoke the utility. The command words used with the INTO FROM phrase not only invoke the utility, but also function as the imperative. Because the command words function as the imperative, and SORT and MERGE are the only command words allowed in this format, you can perform only a SORT or MERGE. You cannot, for example, perform a COPY or a STABLE TAG SORT.
- You cannot use any of the utility's massage features.
- The utility sorts or merges records only in ascending order, based on the records' ASCII collating values.

Examples

You want to merge the transactions of October and November, which are stored in files OCT and NOV, into output file NEW_MASTER. Because you also want OLD_MASTER's records included in the merge process, you want to include OLD_MASTER as an input file. NEW_MASTER already exists and you don't want any statistical output. The command line you need is

```
)MERGE/O/S INTO NEW_MASTER FROM OLD_MASTER OCT NOV)
```

In this next example, you want to sort input file NAMES into output file SORTED_NAMES, which already exists. Instead of suppressing the statistical output, you want to send it to the line printer. The command line you need is

```
)SORT/L=@LPT/O INTO SORTED_NAMES FROM NAMES)
```

Command Line with Command File

You can operate Sort/Merge with only a command line by using the INTO FROM phrase. You can also use both the INTO FROM phrase and a command file together. This combination allows you to name a command file in the command line and to declare

- the output file and all the input files in the command line
- the output file and at least one input file in the command line, and the other input files in the command file
- the output file in the command line and all the input files in the command file

If you declare some input files in the command line and some in the command file, then the utility processes the ones in the command line first.

The formats which allow you this flexibility in declaring input and output files are

SORT /C=filename

{ [{ /L / /L=*filename* }] *[/N] [/O] [/S]* INTO outfile FROM infile ...
[{ /L / /L=*filename* }] *[/N] [/O] [/S]* INTO outfile }

MERGE /C=filename

{ [{ /L / /L=*filename* }] *[/N] [/O] [/S]* INTO outfile FROM infile$_1$ infile$_2$ ...
[{ /L / /L=*filename* }] *[/N] [/O] [/S]* INTO outfile }

/C=filename	directs the utility to use an existing command file, filename.
/L	sends statistical output and any error messages to the current listfile.
/L=filename	sends statistical output and any error messages to *filename* (e.g., an AOS disk file, @ LPT, @ OUTPUT, or @ MTA0). The utility creates an AOS file if it doesn't already exist.
/N	suspends execution of the imperative. The utility still checks the syntax of the command file statements. Thus, you can use /N to speed debugging a command file.
/O	tells the utility that the output file already exists. The utility deletes and then recreates it using the results of the Sort/Merge process.
/S	suppresses the statistical output.
outfile	is the output file.
infile	is an input file.

Command Word and Imperative Relationship

The command file imperative determines which Sort/Merge process the utility performs. The invoking word only invokes the utility.

Examples

You want to repeatedly use the same command file, but you want the utility to write the sorted records to a different output file for each execution. The output file for the first execution is EMPLOYEES and the output file for the second execution is WORKERS. The command file you'll use each time is NAME_SORT which contains these statements:

```
INPUT FILE IS "DEPT_1", RECORDS ARE 70 CHARACTERS.
INPUT FILE IS "DEPT_2", RECORDS ARE 70 CHARACTERS.
KEY 1/12.
SORT.
END.
```

To sort the records in DEPT_1 and DEPT_2 into EMPLOYEES, you need this command line:

```
)SORT/C=NAME_SORT INTO EMPLOYEES↓
```

The utility creates output file EMPLOYEES because it did not exist before. To sort the records into WORKERS, you need this command line:

```
)SORT/C=NAME_SORT INTO WORKERS↓
```

In the prior example, you sorted the same input files into a different output file each execution. In this example, you want to merge different input files into a different output file. For the first execution, the input files are TRANS_1 and TRANS_2, and the output file is MASTER_1. For the second execution, the input files are TRANS_5 and TRANS_6, and the output file is MASTER_2. The command file you'll use each time is TRANS_WITH_TRANS which contains these statements:

```
KEY 1/1.
KEY 50/LAST.
MERGE.
END.
```

To merge TRANS_1 with TRANS_2 into MASTER_1, you need this command line:

```
)MERGE/C=TRANS_WITH_TRANS INTO MASTER_1 FROM TRANS_1 TRANS_2)
```

To merge TRANS_5 with TRANS_6 into MASTER_2, you need this command line:

```
)MERGE/C=TRANS_WITH_TRANS INTO MASTER_2 FROM TRANS_5 TRANS_6)
```

Table 7-1 summarizes all the input and output file declaration options. You use one or more of them if you want to declare some or all of the above files in a command line. The first column of the table divides the options into the two main categories: using the command line with a command file or without. The other two columns tell you what file declaration options are possible in each of the main categories.

Table 7-1. File Declaration Options

Use Command File?	Declare in Command Line	Declare in Command File
yes	none	output and all input
yes	output and all input	none
yes	output only	all input
yes	output and at least one input	other input
no	output and all input	---

End of Chapter

Chapter 8

INFOS II Files

You must use the declarations described in this chapter to define INFOS II files. In a command file declaring INFOS II files, you may use any imperative (described in Chapter 5) and any massage statement (described in Chapter 6).

You can use a command line and no command file to perform limited sorts and merges of non-INFOS II files. With INFOS II files, however, you must use a command file.

INPUT INFOS Declaration

The format of the INPUT INFOS declaration is

```
INPUT INFOS INDEX IS "name"

[ ,PATH IS { DOWN
             *
             "literal"
             "literal":"literal"
             "literal"-
             -"literal"
             Generic Key Selectors** } [,IGNORE LOGICAL DELETES] [,RECORD] [,[TRIMMED] PARTIAL RECORD]

[ ,KEY [ PADDED TO integer CHARACTERS WITH { integer
                                             "literal" } ] ] [,HEADER] { [....] } ]

,RECORDS ARE { integer CHARACTERS
             { VARIABLE UPTO integer CHARACTERS } .
```

**Generic Key Selectors*

```
{ "literal"+
  "literal"+:"literal"
  "literal":"literal"+
  "literal"+:"literal"+
  "literal"+-
  -"literal"+ }
```

You need to visit specific INFOS II keys and extract specific information from each key. The utility uses this information in a Sort/Merge process. If you don't include a PATH IS clause in the INPUT INFOS declaration, then by default the utility visits all the main subindex keys and extracts the records associated with them. To gain more control over which keys the utility visits and what information it extracts from each key, you need a PATH IS clause.

PATH IS Clause

A PATH IS clause consists of at least one *key selector* usually followed by an *extractor phrase*. The key selectors determine which keys in an INFOS II index the utility visits. In other words, the key selectors indicate the path that the utility traverses through an INFOS II index. (A key selector has nothing to do with the key field defined in a KEY declaration.) To tell the utility what information to extract from a given key or set of keys, you must pair a key selector with an extractor phrase. This forms a *key selector/extractor pair*. The paired extractor phrase selects the key, record, and/or partial record that the utility extracts from the visited keys.

We'll now explain all the key selector and extractor phrases. After that, we'll take you through a few examples which show you some ways to combine key selectors and extractor phrases. Both the explanations and the examples refer to Figure 8-1, which shows INFOS II index EXAMPLE. The figure appears twice for your convenience: once near the explanations and once near the examples.

Key Selectors

There are two kinds of key selectors: those referring to an entire subindex level (DOWN and *), and those referring to a key or range of keys in a particular subindex (all the other key selectors).

The level to which a key selector refers depends on where the key selector appears in the INPUT INFOS declaration. We'll have to give you a "sneak preview" of the key selector *, to help you understand the following examples. In these examples, the * tells the utility to visit all the keys at a given subindex level.

If the clause is

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS *,
```

then the * refers to all the keys in the main subindex, or level 0.

If the clause is

```
PATH IS *, *, *,
```

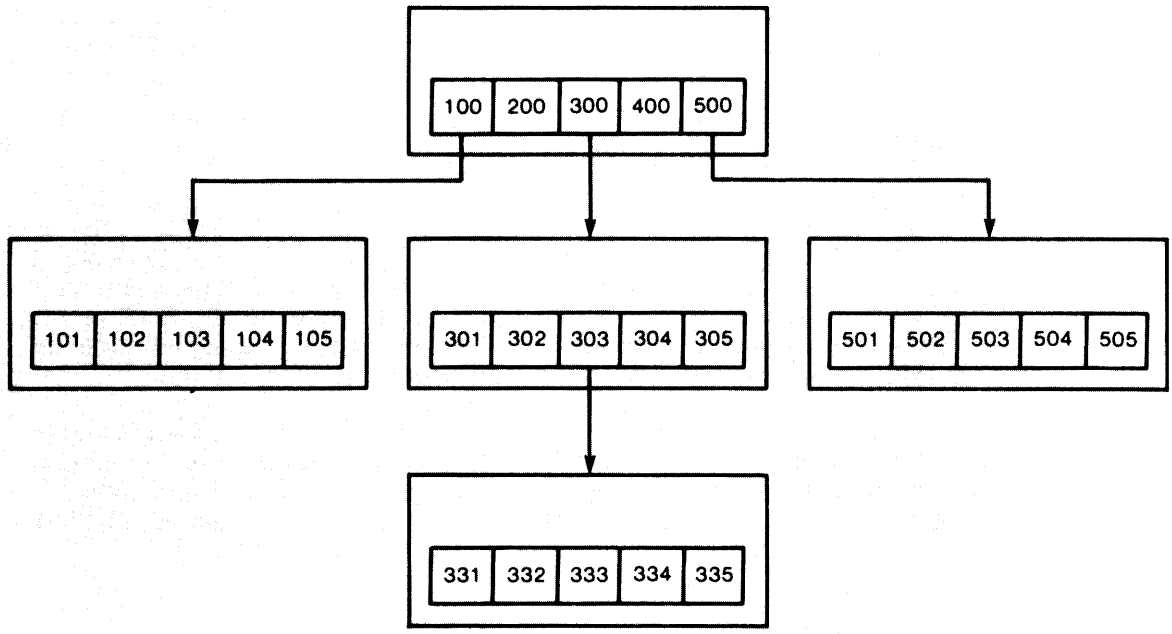
then the first * refers to all the keys in the main subindex, or level 0. The second * refers to all the keys in all the subindexes directly under the main subindex, level 1. The third * refers to all the keys in all the level 2 subindexes.

The clause

```
PATH IS *,
        *,
        *,
```

is also syntactically correct. And it's easier to see which key selector matches which level. We'll use this format for all subsequent examples.

The explanations of the key selectors refer to Figure 8-1. Figure 8-1 represents a simple DBAM index, named EXAMPLE.



SD-02373

Figure 8-1. DBAM Index, EXAMPLE

The key selector phrases and their meanings are

Key Selector	Meaning
DOWN	The utility visits only the first key in the first subindex in a given level. However, Sort/Merge visits this key only for the purpose of finding its linked subindex, which is the first subindex of the next lower level. The utility then travels to that subindex. Therefore, you can think of DOWN in this way: the utility passes through a subindex level without visiting any of its keys. You can link any key to a subindex below it with the INFOS II system. However, because DOWN causes the utility to find the subindex linked to the first key in the first subindex in the level you want to pass through, that key must be linked to a lower level subindex. For example, you can specify PATH IS DOWN, to pass through the main subindex of EXAMPLE, because key 100 is linked to a subindex. But you cannot specify PATH IS DOWN, DOWN, to pass through level 1 because key 101 is not linked to a subindex. It makes no sense to pair DOWN with an extractor phrase since DOWN causes the utility to skip keys rather than visit them. Thus, you use DOWN by itself, and not in a key selector/extractor pair.

* (asterisk)

The utility visits all INFOS II keys in the subindexes that are linked to previously visited keys in the immediately higher subindex level. For example, in the clause

```
PATH IS *,
        *,
```

the first * tells the utility to visit every key in the main subindex. (Sort/Merge always vists the root node, to which the main subindex is attached.) Because it visits every key in level 0, the utility visits keys 100, 300, and 500. Since each of the level 1 subindexes is linked to one of these three keys, the second * tells the utility to visit every key in each subindex in level 1.

Any other key selector which immediately precedes * will limit *'s scope in a given subindex level. We'll precede * with DOWN in this next example. You already saw that DOWN limits the utility to visiting the first key in the first subindex in a given level (only for the purpose of finding out the location of its linked subindex). Thus if you specify

```
PATH IS DOWN,
        *,
```

the utility will not visit keys 300 and 500. Because the utility only visits key 100, *'s scope is limited to the level 1 subindex linked to key 100. In this case, * tells the utility to visit all keys in the subindex linked to key 100.

"literal"

Within a particular subindex, the utility visits only INFOS II key **literal**. For example, the main subindex has these keys: 100, 200, 300, 400, 500. If the clause is

```
PATH IS "300",
```

then the utility visits only key 300.

"literal":"literal"

The utility visits each INFOS II key in the inclusive range between the two literals. That is, it visits both **literal** keys and the keys between them in a particular subindex. The first **literal** must have a lower ASCII value than the second. For example, the main subindex has these keys: 100, 200, 300, 400, 500. If the clause is

```
PATH IS "200":"400",
```

then the utility visits keys 200, 300, and 400.

"literal"-

The utility visits INFOS II key **literal** and each higher (ASCII) value key in a particular subindex. For example, the main subindex has these keys: 100, 200, 300, 400, 500. If the clause is

```
PATH IS "300"-,
```

then the utility visits keys 300, 400, and 500.

-"literal"

The utility visits the lowest (ASCII) value INFOS II key and each higher (ASCII) value key up to and including key literal in a particular subindex. For example, the main subindex has these keys: 100, 200, 300, 400, 500. If the clause is

```
PATH IS -"300",
```

then the utility vists keys 100, 200 and 300.

"literal"+

This is a *generic* key selector. It is useful if you know what records you want to extract, but you don't know their precise keys. A generic key selector tells the utility to locate the first key in a particular subindex that exactly matches literal, up to the length of literal. For example, "AB"+ is two characters long and exactly matches the *leading portion* (here, the first two characters) of INFOS II key ABC. Because generic key selectors are just a subset of key selectors, we won't make this distinction any more (except where appropriate).

The literal must exactly match the leading portion of at least one key in each subindex the utility visits. If you don't meet this condition, you'll get an error message.

For example, suppose you want to visit the first key that begins with the digits 30 in the first level 1 subindex. If you specify this PATH IS clause

```
PATH IS *,
        "30"+,
```

you'll get an error message. This is because no leading portion of keys in the level 1 subindex linked to key 100 exactly matches the key selector "30"+.

To correct the error, you need to exclude the subindex linked to key 100. You do this by changing the PATH IS clause to

```
PATH IS "300",
        "30"+,
```

Key 301 is the first key whose leading portion exactly matches "30"+; thus the utility visits key 301.

You can substitute "literal"+ wherever you use "literal". Thus, by simple substitution we get five more key selectors:

- "literal"+:"literal"
- "literal":"literal"+
- "literal"+:"literal"+
- "literal"+-
- -"literal"+

Table 8-1 shows an example for each of these key selectors.

Table 8-1. (Generic) Key Selector Examples

Key Selector	Example	Utility Visits
"literal"+:"literal"	"2"+:"400"	keys 200, 300, and 400
"literal":"literal"+	"200":"4"+	keys 200, 300, and 400
"literal"+:"literal"+	"2"+:"4"+	keys 200, 300, and 400
"literal"+-	"2"+-	keys 200, 300, 400, and 500
-"literal"+	-"4"+	keys 100, 200, 300, and 400

Extractor Phrases

The extractors tell the utility what information to extract when it visits a key. Any combination of extractors forms an *extractor phrase*. You may pair an extractor phrase with each unique key selector except DOWN. You aren't required to use an extractor phrase with a key selector; you can use a key selector by itself. In that case, the utility visits a key without extracting any information associated with that key. The extractors and the action they take are

Extractor **Action**

RECORD extracts the record associated with each key that RECORD's paired key selector tells the utility to visit. For example, the PATH IS clause

PATH IS *, RECORD,

tells the utility to extract the record associated with each key in the main subindex.

PARTIAL RECORD extracts the partial record asociated with each key that PARTIAL RECORD's paired key selector tells the utility to visit. For example, the PATH IS clause

PATH IS *, PARTIAL RECORD,

tells the utility to extract the partial record associated with each key in the main subindex.

If you specify PARTIAL RECORD TRIMMED, then the utility strips all trailing nulls from the partial records it extracts.

KEY extracts the INFOS II key(s) that KEY's paired key selector tells the utility to visit. If you include the PADDED TO option, the utility pads each key it extracts to integer characters. The pad character is literal or integer, the decimal equivalent of an ASCII character.

For example, the PATH IS clause

PATH IS *, KEY PADDED TO 6 CHARACTERS WITH "-",

extracts all the keys in the main subindex. And it pads all the keys to 6 characters with - (dash). For instance, the utility pads key 100 with - (dash) until the key is 6 characters long: 100---.

Extractor	Action
HEADER	generates a two-word, nonprinting binary header. It does this for each INFOS II key that HEADER's paired key selector tells the utility to visit. The first word of the header contains the length of the record associated with the INFOS II key. The first byte of the second word contains the partial record length for that key; the second byte of the second word contains the length of the key.

IGNORE LOGICAL DELETES

IGNORE LOGICAL DELETES further qualifies what information the utility extracts. It causes the utility to ignore extractors paired with a key selector whenever that key selector tells the utility to visit a logically deleted key. In these cases, the utility visits logically deleted keys but doesn't extract any information. (Sort/Merge considers a key logically deleted if either INFOS KEY ("LOCAL") or INFOS RECORD ("GLOBAL") LOGICAL DELETE is in effect.)

If you use this phrase, you must place it immediately after a key selector. Thus, if you want to use this phrase with a key selector/extractor pair, you must place IGNORE LOGICAL DELETES between the key selector and extractor phrase. For example:

```
PATH IS *, RECORD,
        *, IGNORE LOGICAL DELETES, RECORD,
```

RECORD is the extractor phrase. In this case, the utility visits all level 1 keys but doesn't extract the records associated with logically deleted keys.

RECORDS ARE Clause

The INPUT INFOS declaration always requires the RECORDS ARE clause. If the database associated with the index you named contains fixed-length records, use the integer CHARACTERS phrase. For example:

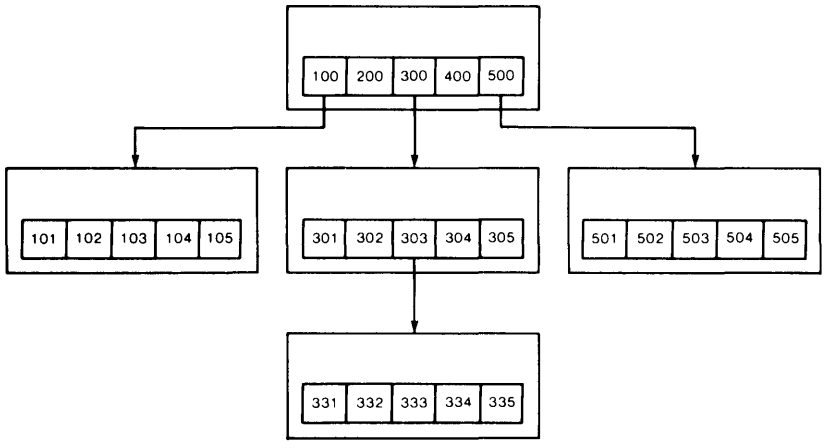
```
RECORDS ARE 70 CHARACTERS
```

If the database contains variable-length records, use the VARIABLE UPTO phrase. For example:

```
RECORDS ARE VARIABLE UPTO 100 CHARACTERS
```

Examples

All the examples in this section refer to Figure 8-1, repeated here for your convenience.



Each example shows you the complete INPUT INFOS declaration you need to accomplish various tasks. We'll carefully dissect each example's PATH IS clause. Assume that EXAMPLE's database contains variable records not longer than 100 characters each.

If you want to perform a preordered traversal of index EXAMPLE, extracting the record associated with each key visited, specify

```
INPUT INFOS INDEX IS "EXAMPLE".
        PATH IS *, RECORD,
                  *, RECORD,
                  *, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

In the PATH IS clause, the first key selector (*) tells the utility to visit each key in the main subindex. The first extractor (RECORD) tells the utility to extract the record associated with each key in the main subindex. The next key selector/extractor pair (*, RECORD) tells the utility to visit every key in every subindex at level 1, and extract the records associated with those keys. The last key selector/extractor pair indicates the same procedure for all level 2 subindexes (there is only one).

In a preordered traversal of this index, the utility visits the keys in the following order (an arrow indicates a change in subindex level):

100 → 101, 102, 103, 104, 105 → 200, 300 → 301, 302, 303
→ 331, 332, 333, 334, 335 → 304, 305 → 400, 500
→ 501, 502, 503, 504, 505

Suppose that you logically deleted key 303's record. Because we didn't qualify the key selector * with IGNORE LOGICAL DELETES, the utility will still visit key 303 and its associated keys in the lower level subindex.

If you don't want to extract information from logically deleted key 303, then specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS *, RECORD,
                  *, IGNORE LOGICAL DELETES, RECORD,
                  *, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The third key selector/extractor pair tells the utility to visit all the keys in the level 2 subindex and to extract the associated records.

Now the utility visits the keys in the following order:

100 → 101, 102, 103, 104, 105 → 200, 300 → 301, 302
→ 331, 332, 333, 334, 335 → 304, 305 → 400, 500
→ 501, 502, 503, 504, 505

Suppose that there are logically deleted keys in index EXAMPLE you don't want to vist, but you don't know which keys they are. You could specify the following in order to avoid keys associated with logically deleted records, wherever they may occur:

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS *, IGNORE LOGICAL DELETES, RECORD,
                  *, IGNORE LOGICAL DELETES, RECORD,
                  *, IGNORE LOGICAL DELETES, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

If you want to extract the keys in the subindex linked to key 500, and pad them to 10 characters with the plus sign (+), specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS "500",
                *, KEY PADDED TO 10 CHARACTERS WITH "+",
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The first key selector ("500") is not paired with an extractor. This tells the utility to visit key 500 but not to extract any information associated with it. The second key selector (*) tells the utility to visit every key in the subindex linked to key 500. Sort/Merge doesn't visit the other level 1 subindexes because a "literal" key selector ("500") immediately precedes this *, restricting its scope. The *'s paired extractor phrase (KEY PADDED TO 10 CHARACTERS WITH "+") tells the utility to extract each key in the subindex, and pad them with + to a fixed-length of 10 characters. For example, key 505 is extracted as 505+++++++.

If you want to extract the keys, partial records, and records from all keys in the level 1 subindexes, specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS *,
                *, KEY, PARTIAL RECORD, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

In order for the utility to visit all the level 1 subindexes, it must first visit all the keys at the immediately higher level to which these indexes are linked. The first * tells the utility to visit all keys in the main subindex; thus, the utility visits the keys to which the level 1 indexes are attached. Also, the first key selector (*) is not paired with an extractor phrase. This tells the utility not to extract any information from the main subindex keys.

The second key selector (*) tells the utility to visit every key in every level 1 subindex. Its paired extractor phrase (KEY, PARTIAL RECORD, RECORD) tells the utility to extract the key, partial record, and associated record, from each key it visits.

If you want to extract the records from only the level 2 subindex keys, specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS "300",
                "303",
                *, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS
```

The first key selector ("300"), a literal, tells the utility to visit only key 300 in the main subindex. The utility doesn't extract any information associated with key 300, since this key selector isn't paired with an extractor phrase. The next key selector ("303"), another literal, tells the utility to visit key 303. Because this key selector also is not paired with an extractor phrase, the utility doesn't extract any information associated with key 303. The key selector/extractor phrase pair *, RECORD tells the utility to visit every key in the subindex linked to key 303 and to extract their records.

If you want to extract the records for the first three keys in the subindex linked to key 100, specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS DOWN,
                -"103", RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The key selector DOWN tells the utility to pass through the main subindex without visiting any of its keys (except the first). The utility then travels to the subindex linked to key 100 because 100 is the lowest (ASCII) value key in the main subindex. (Thus in this case, DOWN is equivalent to "100" without a paired extractor phrase). The key selector -"103" tells the utility to visit key 103 and all the keys in this subindex whose value is less than 103. RECORD, the paired extractor phrase, tells the utility to extract the records associated with the keys it visits.

If you want to extract the records from the last three keys in the subindex linked to key 500, specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS "500",
                "503"-, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The key selector "500" tells the utility to visit key 500. The utility extracts no information from this key because it isn't paired with an extractor phrase. The key selector "503"- tells the utility to visit key 503 and all higher (ASCII) value keys in the subindex linked to key 500. The paired extractor phrase, RECORD, tells the utility to extract the records associated with the keys it visits.

If you want to extract the partial records from the keys 332, 333, and 334, specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS "300",
                "303",
                "332":"334", PARTIAL RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The key selectors "300" and "303" define a path through the index to the level 2 subindex. The key selector "332":"334" tells the utility to visit the keys in the inclusive range of 332 to 334. PARTIAL RECORD, the paired extractor phrase, tells the utility to extract the partial record associated with the keys it visits.

If you want to use generic key selectors to extract all the records from keys whose first digit is 3, you could specify

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS "3"+, RECORD,
                "3"+-, RECORD,
                "3"+-, RECORD,
        RECORDS ARE VARIABLE UPTO 100 CHARACTERS.
```

The utility extracts the record from each key it visits. The key selector "3"+ tells the utility to visit key 300. The subindex level 1 key selector "3"+- tells the utility to visit keys 301 through 305. The subindex level 2 key selector "3"+- tells the utility to visit keys 331 through 335.

Extracted Information

In all these examples, you extracted one or more types of information (record, partial record, and/or key) from an INFOS II file. What happens to this information after you extract it? The answer to this question is relevant now, and will be important when we discuss the OUTPUT INFOS declaration.

After each visit, the utility extracts the desired information, strings it altogether, and treats it as one record. For example, suppose key 100's record is abcdefg, and its partial record is abc. If we extract the key, record, and partial record from key 100, in that order, then the utility strings together 100, abcdefg, and abc, producing the record 100abcdefgabc.

You can send this record directly to an AOS file. Let's return to INFOS II index file EXAMPLE for an example of how you could do this. Suppose that key 200's record is hijklmn, and its partial record is hij. The record the utility creates when it extracts the key, record, and partial record after it visits key 200 is 200hijklmnhij. The other records formed by concatenation are similar to this record (for example, 300opqrstuopq, and so on).

By declaring AOS file TEMP in an AOS OUTPUT FILE declaration, you can sort the records from INFOS II file EXAMPLE into AOS file TEMP. You could base the sort on the first three character positions of the records (the INFOS II key), and arrange them in descending order. The command file you need to accomplish this is

```
INPUT INFOS INDEX IS "EXAMPLE",
        PATH IS *, KEY, RECORD, PARTIAL RECORD,
                *, KEY, RECORD, PARTIAL RECORD,
                *, KEY, RECORD, PARTIAL RECORD,
        RECORDS ARE VARIABLE UPTO 50 CHARACTERS.
OUTPUT FILE IS "TEMP", RECORDS ARE VARIABLE UPTO 50 CHARACTERS.
KEY 1/3 DESCENDING.
SORT.
END.
```

You can also send the records formed by concatenation to an INFOS II output file. Whether you send the entire record or only part of the record to an INFOS II file depends on the OUTPUT INFOS declaration. The utility can treat all or part of each record as a key, partial record, or record, again depending on the OUTPUT INFOS declaration. The next section discusses these possibilities and the OUTPUT INFOS declaration in detail.

OUTPUT INFOS Declaration

Sort/Merge cannot create INFOS II files. Therefore, any INFOS II output file that you name must exist before you invoke the utility.

The format of the OUTPUT INFOS declaration is

```
OUTPUT INFOS { INDEX     } IS "name"
             { INVERSION }
[
  [, RECORD IS { integer/integer } ] [, PARTIAL RECORD IS { integer/integer } ]
               { integer/LAST    }                        { integer/LAST    }
  [, TRIM [ { integer   } FROM ] KEYS ]
          { "literal" }
  [, PATH IS [ { down      } , [...] ] * ] ]
             { "literal"   }
             { "literal"+  }
] .
```

OUTPUT INFOS IS Clause

The OUTPUT INFOS clause requires that you choose between one of two options, INDEX or INVERSION. Inversion occurs when two different INFOS II keys are linked to the same database record. In general, if you don't plan to invert keys, use the INDEX option; if you plan to invert keys, use the INVERSION option.

Figure 8-2 shows a case where you use the INDEX option because you don't invert keys. The first half of the figure shows an INFOS II file before you execute a command file. The subindex linked to KEY2 exists, but has no INFOS keys in it. The second half of the figure shows the INFOS II file after you execute a command file. The INFOS II file now has a new key, NEW, in its previously empty subindex. And it has a new record, NEW RECORD, in its database. The blue line represents the linkage made between NEW and NEW RECORD. Note that keys NEW and OLD are linked to different records; they are not inverted.

Figure 8-3 shows a case where you use the INVERSION option because you do invert keys. The first half of the figure shows an INFOS II file before you execute a command file. The subindex linked to KEY2 exists, but has no INFOS II keys in it. The second half of the figure shows the INFOS II file after you execute a command file. The INFOS II file now has a new key, NEW, in its previously empty subindex. But it doesn't have a new record in its database; instead, you linked NEW to SAME RECORD, the record to which key OLD is already linked. The blue line represents the linkage made between NEW and SAME RECORD. Because keys NEW and OLD are linked to the same record, they are inverted.

RECORD IS and PARTIAL RECORD IS Clauses

Recall that in the section "Extracted Information," we discussed how Sort/Merge creates a record from extracted information. For example, say that the utility extracts the record and partial record from a given key of an INFOS II input file. The utility then creates a record which is the concatenation of the record and the partial record.

The RECORD IS clause determines which portion of this record will become a database record associated with an INFOS II key in an INFOS II output file. The PARTIAL RECORD IS clause determines which portion of this record will become a partial record associated with an INFOS II key in an INFOS II output file. Each clause's location phrase defines the range of characters that will become a record or partial record. Let's go through an example slowly. The example refers to Figures 8-4 to 8-6.

Figure 8-4 shows INFOS II index A, whose first key is 100. Key 100's partial record is 123. Key 100 is linked to database record ABCDEFG. You want the utility to visit key 100 and extract its record and partial record. Afterwards, the utility concatenates 123 and ABCDEFG to form the concatenated record ABCDEFG123 (also shown in Figure 8-4). The utility uses this record in a Sort/Merge process. In this example, you want the utility to copy portions of the concatenated record to another INFOS II index, named B.

You want to define character positions 1 through 3 of the concatenated record to be the partial record copied to B. And you want to define character positions 4 to 10 of this same record to be the record copied to B. Figure 8-5 shows the concatenated record divided into a partial record and a record. To accomplish this, you need the following clauses:

```
RECORD IS 4/10, PARTIAL RECORD IS 1/3
```

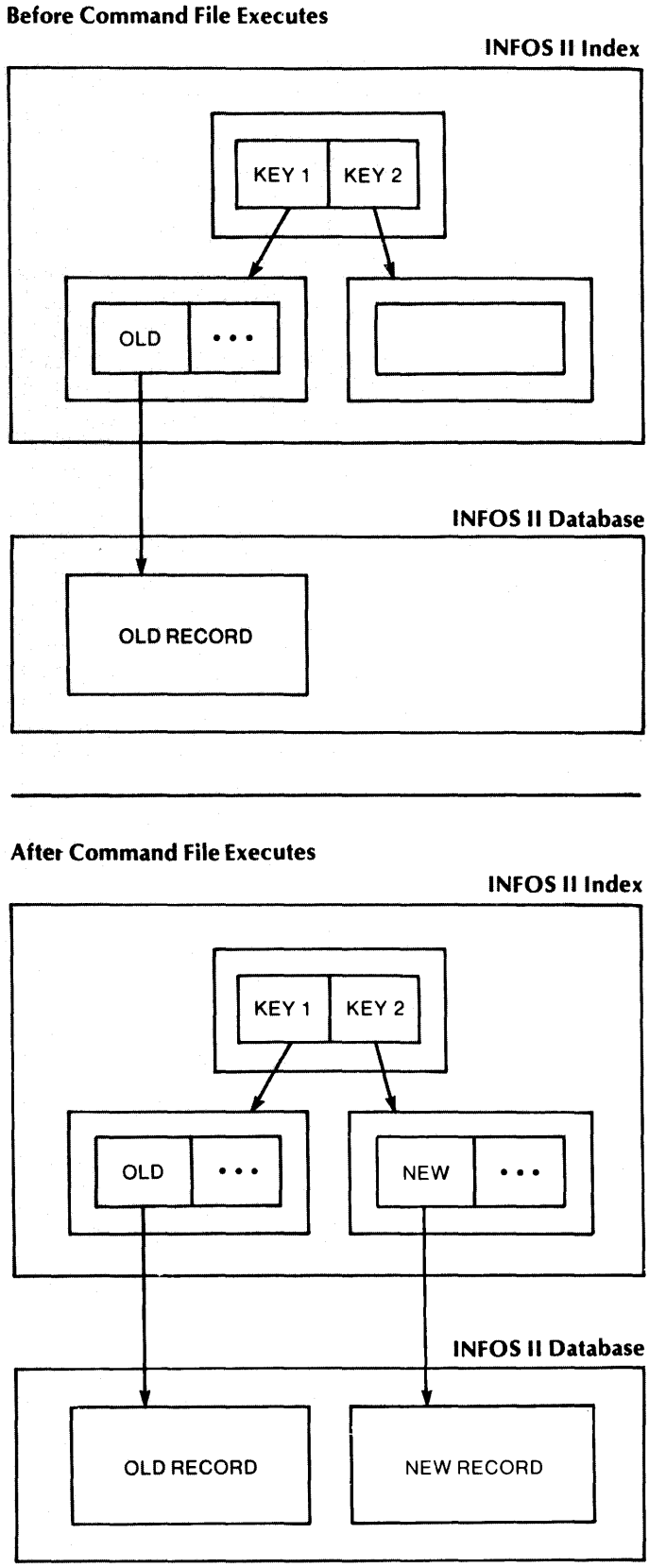


Figure 8-2. INDEX Option Case

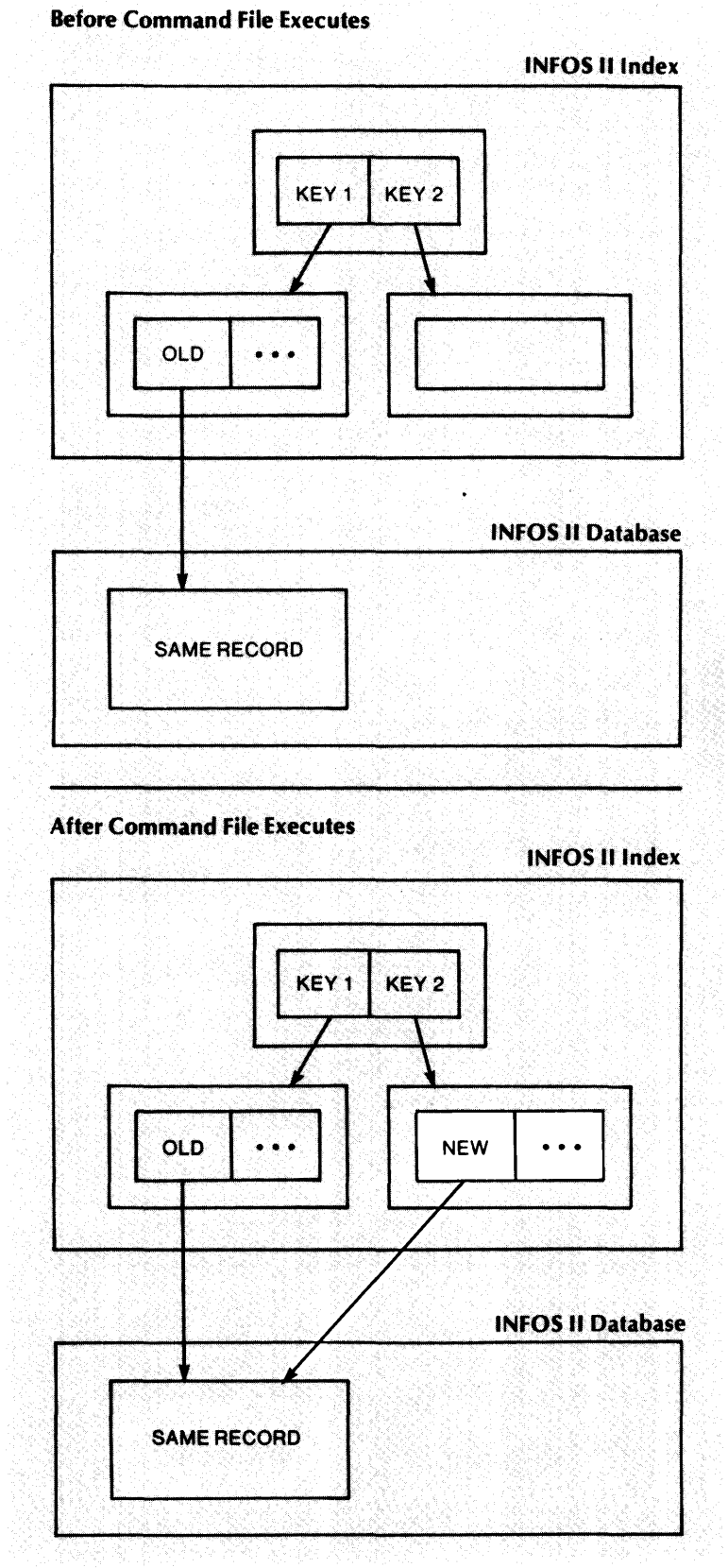


Figure 8-3. INVERSION Option Case

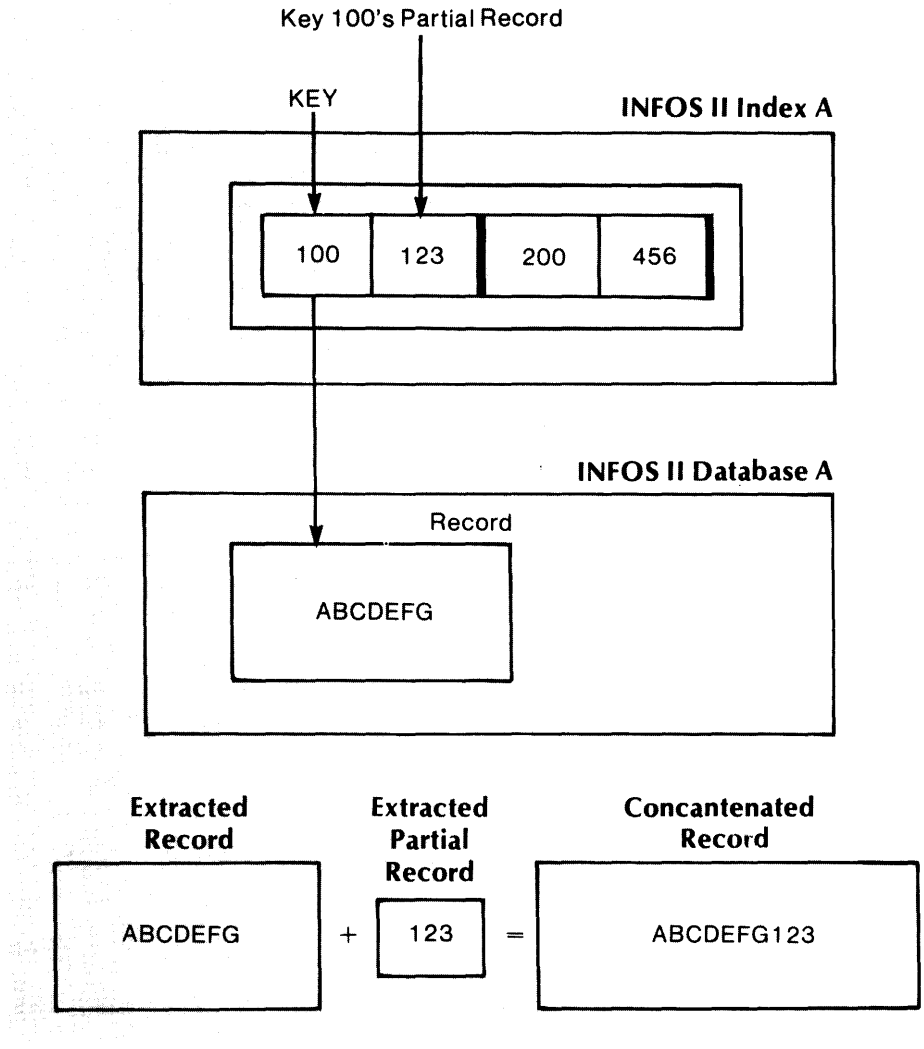


Figure 8-4. Record Formed by Concatenation

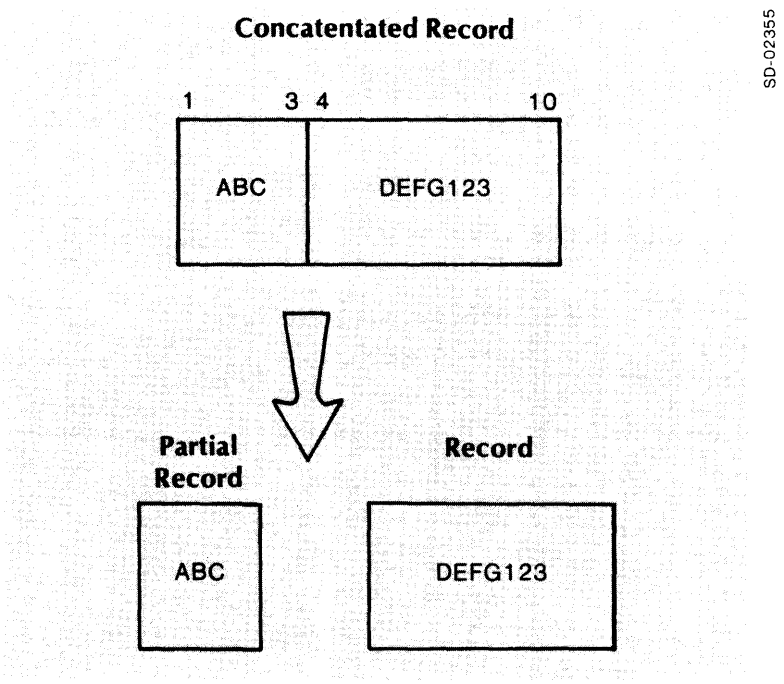


Figure 8-5. Partial Record and Record from Concatenated Record

Figure 8-6 shows where the partial record and the record end up after the utility sends them to INFOS II index B (and its associated database). How did the utility know where to put them? We'll answer this question later.

The last example shows how the utility can divide a record created from the concatenation of information extracted from an INFOS II file. The utility divides records from AOS files in exactly the same way. If ABCDEFG123 were a record in an AOS file, you would use the same clause:

```
RECORD IS 4/10, PARTIAL RECORD IS 1/3
```

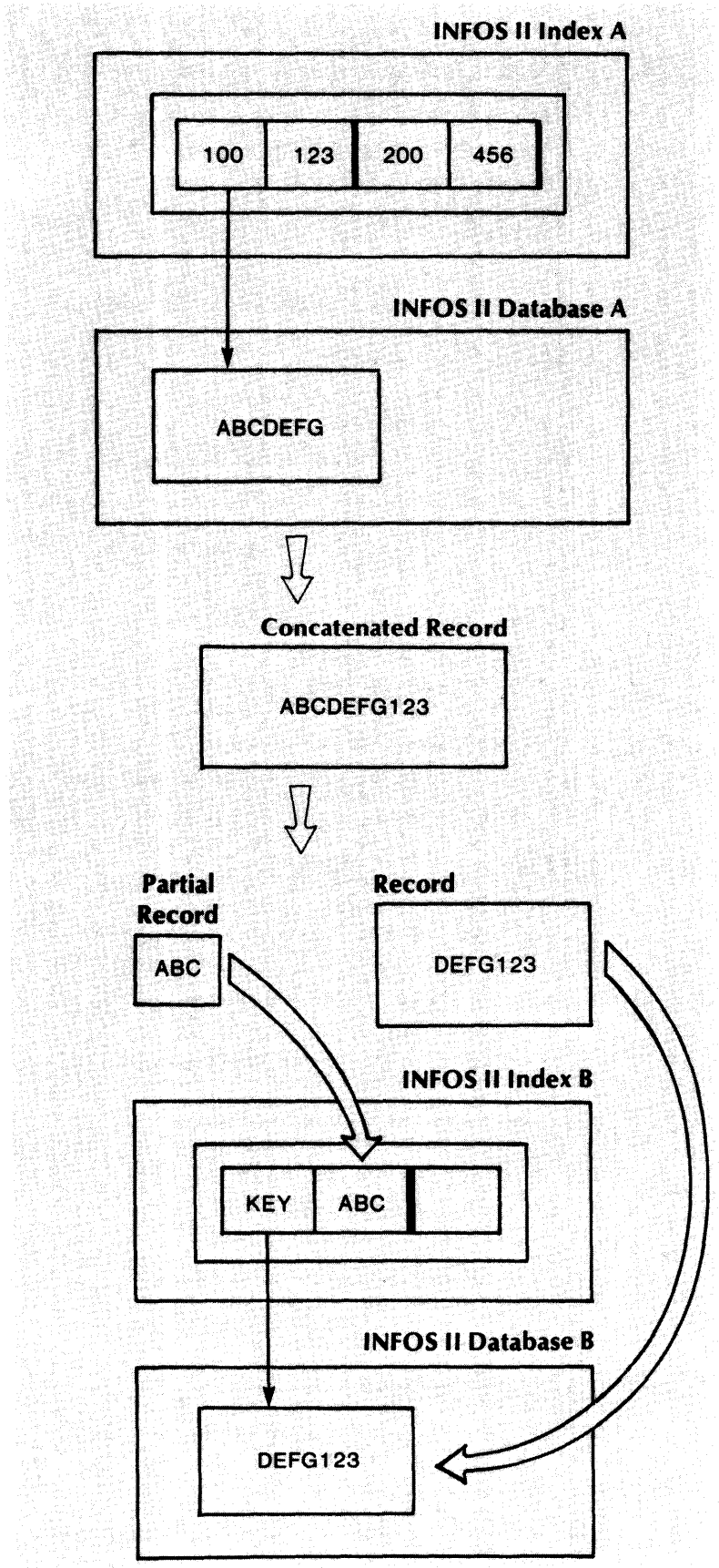
to divide the record the way you want. It doesn't matter whether the record that the utility "dissects" comes from an INFOS II or AOS file.

The ranges for the record and the partial record, defined in the concatenated record, can overlap. For example, the entire concatenated record can become the record copied to INFOS II index B's database, and character positions 4 to 6 can become a key's partial record in INFOS II index B. To accomplish this, you specify the following clauses:

```
RECORD IS 1/LAST, PARTIAL RECORD IS 4/6
```

These clauses make ABCDEFG123 the record and DEF the partial record.

So far we've discussed dividing a concatenation of information into partial records and records. However, if the utility extracts only a record, only a partial record, or only a key from an INFOS II key in an INFOS II input file, the utility doesn't concatenate information. For example, suppose that the utility extracts only database records. These database records, which are not formed by concatenation, become the records that Sort/Merge processes. And in these records, the RECORD IS and PARTIAL RECORD IS clauses define ranges. Figure 8-7 first shows a record which the utility extracts from INFOS II index A's database (after visiting the ALPHA CO. key). It then shows the database record divided into a record and a partial record. Finally, it shows where this information ends up in INFOS II index B and its database.



SD-02356

Figure 8-6. Information Sent from A to B

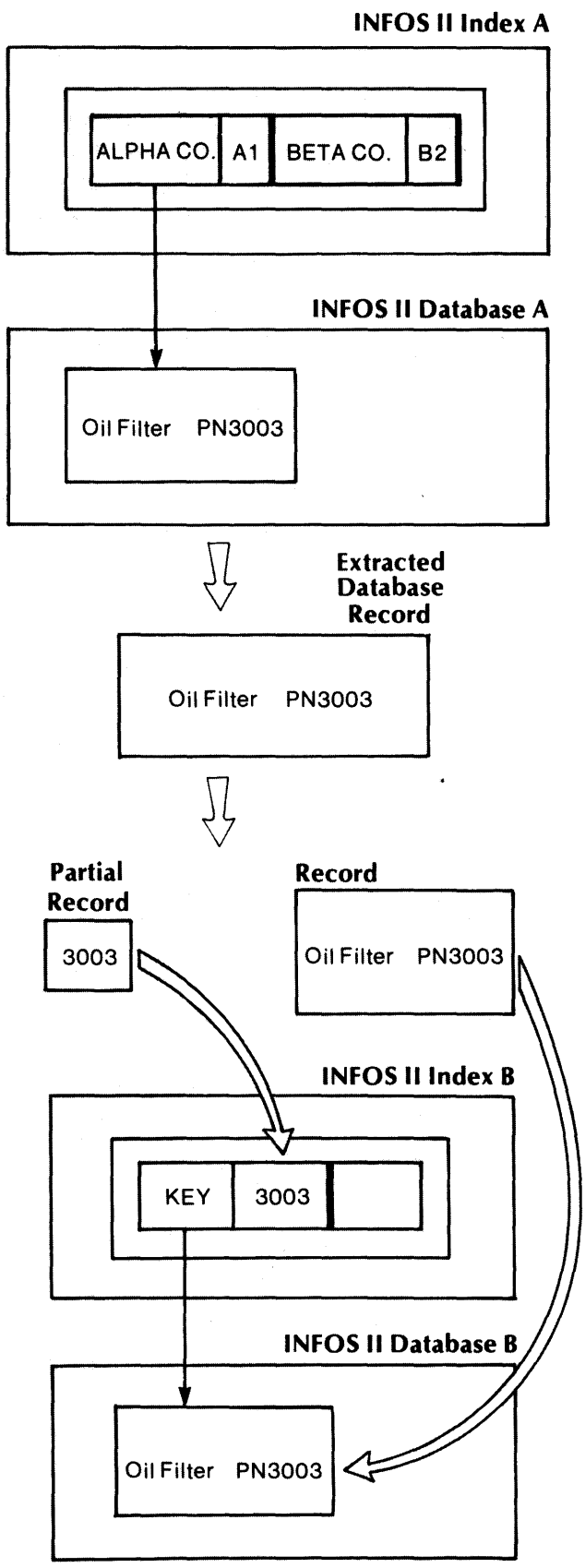


Figure 8-7. Database Record Becomes Record and Partial Record

Defining the INFOS II Key

We now discuss a potentially confusing point, and therefore ask the reader to pay special attention. You saw how the utility portions a record into a record and partial record, and then how it sends this information to an INFOS II output file. How do you define an INFOS II key from a record?

The syntax of the OUTPUT INFOS declaration does not allow you to define a key in an input file record. You must instead use the KEY declaration to define an INFOS II key. Thus, we're using the KEY declaration to define the output INFOS II key location as well as the field the utlity uses for sorting or merging.

You can define an INFOS II key without defining a partial record or record. In other words, you can use a KEY declaration to define an INFOS II key without including a RECORD IS or PARTIAL RECORD IS clause in the INFOS OUTPUT declaration. This is essentially the same as performing an INFOS II Write operation in which you specify the Suppress Database Access option. (The *INFOS II System User's Manual* describes this option.)

You don't have to worry about sorting keys into an INFOS II output file, since INFOS II automatically sorts keys (in ascending order) for you. Thus, you would only copy or merge keys (and records and partial records) from one or more INFOS II files to another.

You define an INFOS II key from the same record that you define a record and/or partial record. For example, assume that the utility extracts the database record shown in Figure 8-8.

You want to make character positions 15 to 18 (3003) an INFOS II key, as shown in Figure 8-9. This requires the following KEY declaration:

```
KEY 15/18.
```

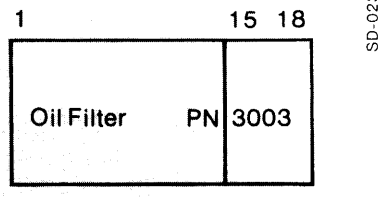


Figure 8-8. Database Record

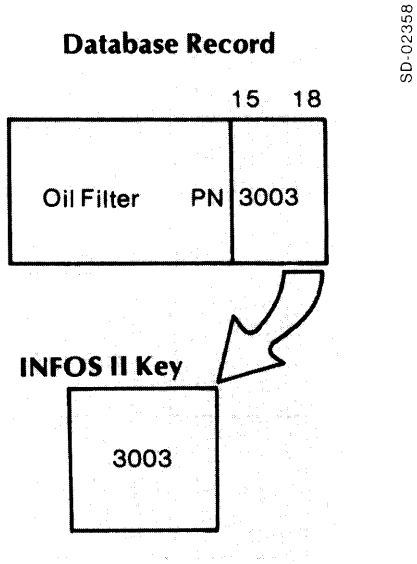


Figure 8-9. KEY Declaration Defines INFOS II Key

Just as the character positions of the record and partial record can overlap, the character positions of the INFOS II key, the record, and/or the partial record can overlap. For example, suppose that the utility extracts the database record shown in Figure 8-10.

You want to make character postions 1 through 25 (the entire record) the record, positions 14 to 15 (A1) the partial record, and positions 22 to 25 (3003) the key. Figure 8-11 shows the database record divided in this way.

The RECORD IS and PARTIAL RECORD IS clauses, and KEY declaration that you need to accomplish this are

```
RECORD IS 1/LAST, PARTIAL RECORD IS 14/15
...

KEY 22/25.
```

You define an INFOS II key from an AOS record the same way that you define an INFOS II key from a record originating from an INFOS II file. As we noted before, it doesn't matter whether the record that the utility dissects comes from an INFOS II or AOS file.

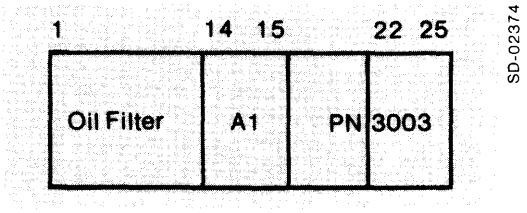


Figure 8-10. Database Record

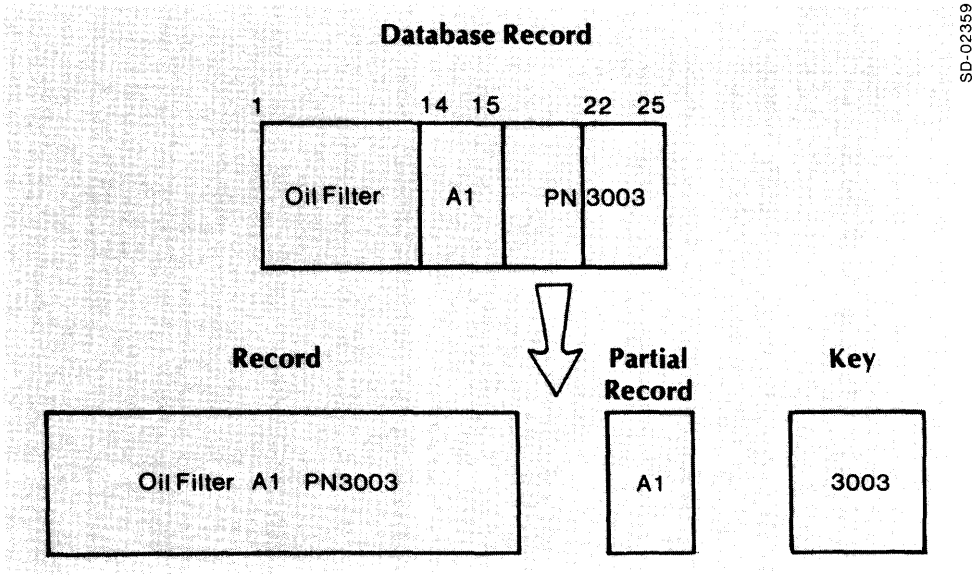


Figure 8-11. Defining Record, Partial Record, and Key from Database Record

TRIM KEYS Phrase

The TRIM KEYS phrase deletes a specific character from the end of each INFOS II key before the utility sends the keys to an INFOS II output file. The default literal for this phrase is the null character. Thus, specifying

```
TRIM KEYS
```

is the same as specifying

```
TRIM "<000>" FROM KEYS
```

The utility will then trim all trailing nulls from the INFOS II keys.

Since the INFOS II utility does not allow zero length keys, Sort/Merge will never trim a key to less than one character.

PATH IS Clause

The RECORD IS and PARTIAL RECORD IS clauses, and the KEY declaration, define *what* the utility will send to the INFOS II output file. The PATH IS clause determines *where* this information goes in the file.

The key selectors DOWN, *, "literal" and "literal"+ have the same meanings as in the INPUT INFOS declaration's PATH IS clause. However, there is one difference in how you list these key selectors. In the INPUT INFOS declaration, you can list these key selectors in any order (given that the INFOS II index file is appropriately structured). In the OUTPUT INFOS declaration, you must specify * as the last key selector in the list. And you may not specify * anywhere else. Because * must be last, the utility can write data to only one subindex each time that you execute a command file.

You would never pair an OUTPUT INFOS declaration key selector with an extractor; the information that you want is already extracted. Again, an OUTPUT INFOS declaration's key selectors determine where the the partial records and INFOS II keys will go in an INFOS II index.

The PATH IS clause is optional. By default, not including the PATH IS clause is the same as specifying

```
PATH IS *.
```

If you omit both the PATH IS and RECORD IS clauses, the utility assumes that you want

```
RECORD IS 1/LAST, PATH IS *.
```

However, if you use the PATH IS clause and want to write records, you must also explicitly use the RECORD IS clause; the utility will not automatically take the default RECORD IS clause.

Examples

Figure 8-12 shows DBAM file MASTER, on which we base the next example. MASTER's main index contains three keys: a part number key (PN), a purchase order key (PO), and a vendor name key (VENDOR). The subindexes to which PN and PO are linked both exist and have keys. The subindex to which VENDOR is linked exists (Sort/Merge can't create a subindex), but has no keys.

Let's say that you have a previously sorted AOS sequential file named TEMP containing vendor records. Figure 8-13 shows the records' format.

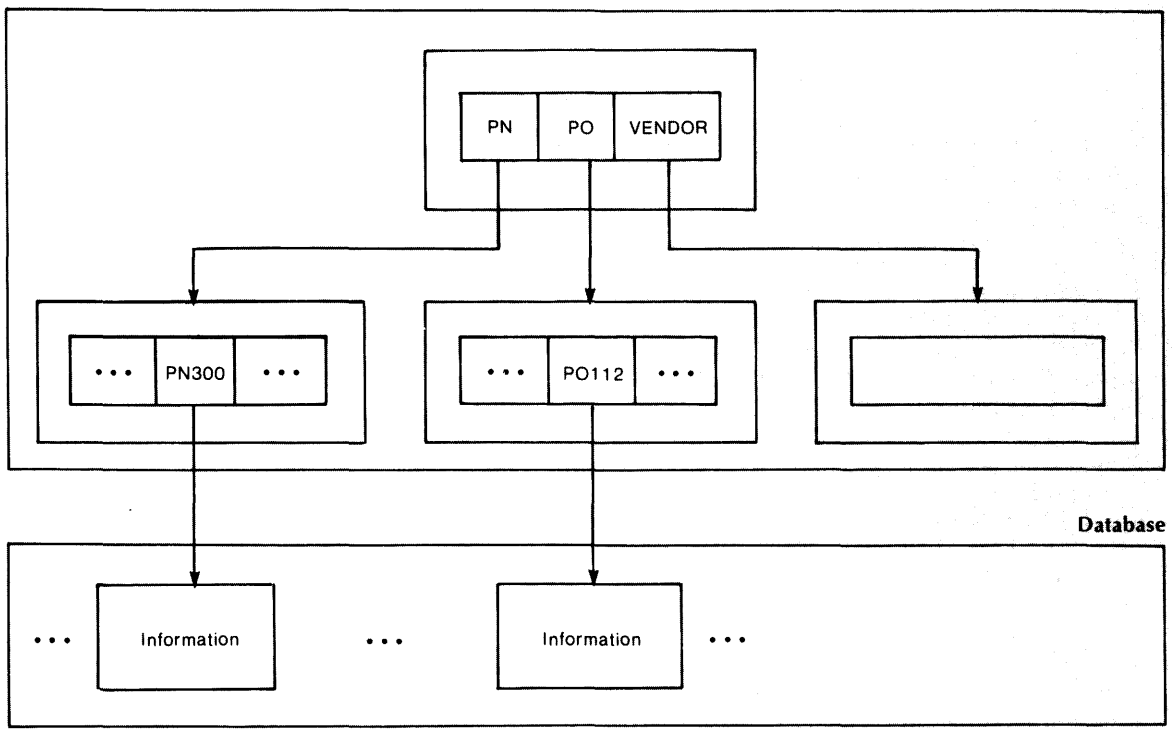


Figure 8-12. MASTER, before Command File Executes

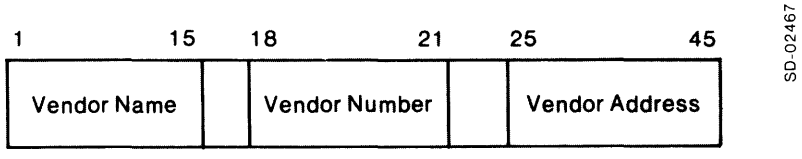


Figure 8-13. Format of TEMP's Records

You want to

- copy the records to MASTER'S database
- make the Vendor Name field the INFOS II key, and copy it to MASTER's VENDOR subindex
- make the Vendor Number field the partial record, and copy it to MASTER's VENDOR subindex

To accomplish this, here's the command file you'll need:

```
INPUT FILE IS "TEMP", RECORDS ARE 45 CHARACTERS.
OUTPUT INFOS INDEX IS "MASTER",
        RECORD IS 1/LAST,
        PARTIAL RECORD IS 18/21,
        PATH IS "VENDOR",
                *.
KEY 1/15.
COPY.
END.
```

Now we'll carefully analyze the command file. We declare TEMP, the input file, in an AOS INPUT FILE declaration. The OUTPUT INFOS declaration uses the INDEX option because you don't want to link keys in the VENDOR subindex to records already existing in MASTER's database. Instead, you want to link keys in the VENDOR subindex to new database records. The RECORD IS clause determines which portion of each of file TEMP's records will be a new record for MASTER's database. Because the RECORD IS clauses's location phrase is 1/LAST, each entire record in TEMP becomes a new record for the database.

Let's leave the OUTPUT INFOS declaration for a moment and look at the KEY declaration. The KEY declaration determines which portion of each of TEMP's records will be an INFOS II key for the VENDORS subindex. You want the Vendor Name field in each record as the INFOS II keys in this subindex. Thus, you define the location phrase of the KEY declaration as 1/15, the character positions of the Vendor Name field.

Now let's return to the OUTPUT INFOS declaration. The PARTIAL RECORD IS clause determines which portion of each of TEMP's records will become a partial record associated with a an INFOS II key in the VENDOR subindex. You want the Vendor Number field (character positions 18 to 21) to be the INFOS II keys' partial records. Thus, the location phrase of the PARTIAL RECORD IS clause is 18/21.

Let's summarize the major points we made about this command file. The RECORD IS and PARTIAL RECORD IS clauses, and the KEY declaration apportion TEMP's records. The PATH IS clause determines where in MASTER the INFOS II keys go. To these keys, the partial records are attached and the database records are linked. Here, the PATH IS clause tells the utility to place the INFOS II keys in the subindex linked to VENDOR.

Figure 8-14 shows what MASTER and its associated database file look like after you execute the command file.

The next example works with a DBAM file of employee records, named EMPLOYEE, shown in Figure 8-15. As it stands, EMPLOYEE has two subindexes: one for employee name keys, and one for employee number keys. The NAME subindex exists but has no keys. The database records linked to the NUMBER keys contain both employee names and numbers. Your goal is to extract the employee last names from the database records, turn them into keys for the NAME subindex, and link those keys to the existing database records.

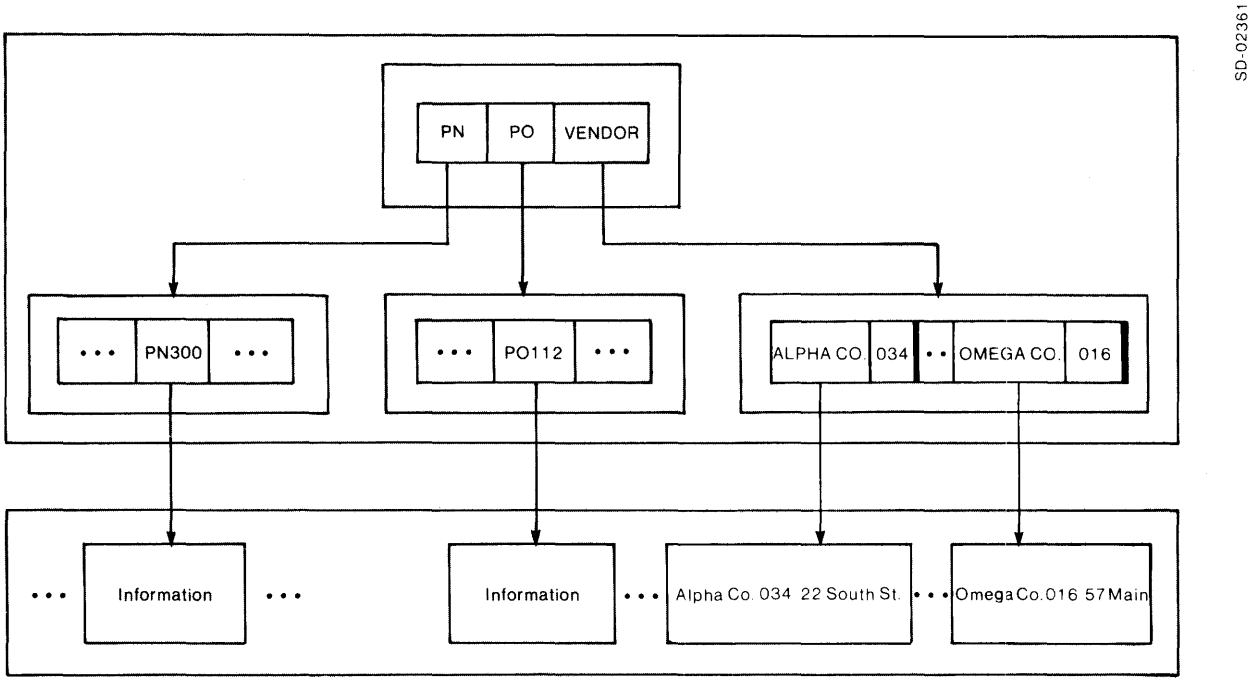


Figure 8-14. MASTER, after Command File Executes

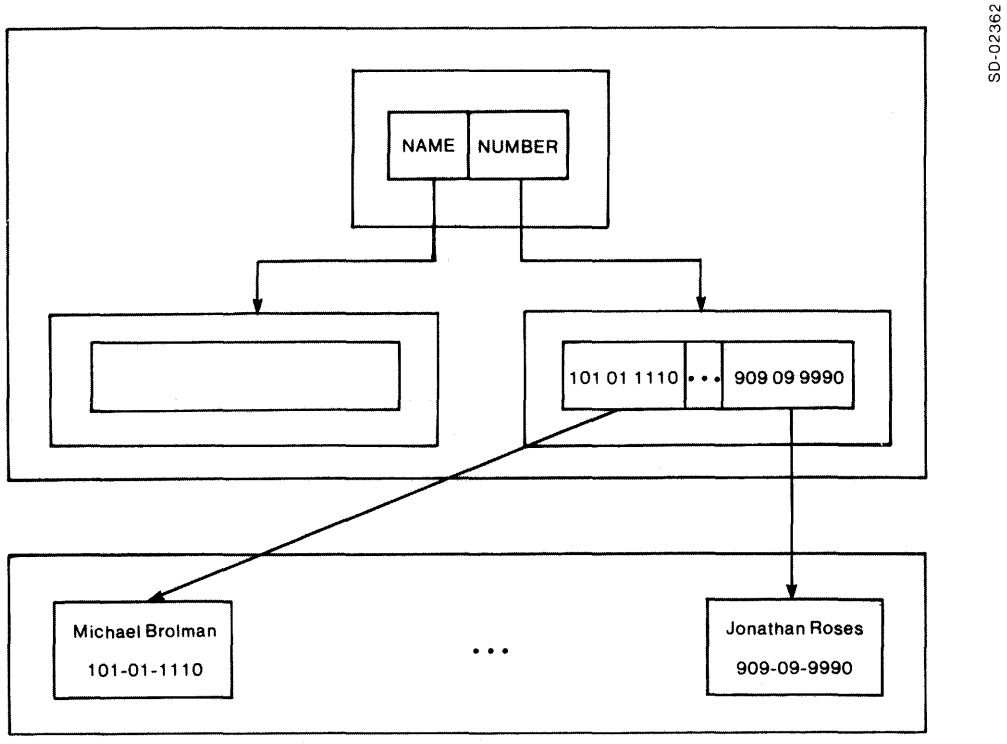


Figure 8-15. EMPLOYEE, before Command File Executes

Figure 8-16 shows the database records' format.

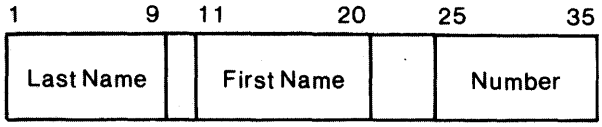


Figure 8-16. Format of EMPLOYEE's Database Records

The command file you need is

```
INPUT INFOS INDEX IS "EMPLOYEE",
        PATH IS "NUMBER",
                *, RECORD,
        RECORDS ARE 35 CHARACTERS.
OUTPUT INFOS INVERSION IS "EMPLOYEE",
        PATH IS "NAME",
                *.
KEY 1/9.
COPY.
END.
```

Now we'll analyze the command file. You want to visit all the keys in the NUMBER subindex so that you can extract all the database records. The PATH IS clause in the INPUT INFOS declaration accomplishes this. The OUTPUT INFOS declaration uses the INVERSION option because you eventually want keys in the NAME subindex linked to the same database records to which the keys in the NUMBER subindex are linked. The PATH IS clause tells the utility to put the keys in the NAME subindex. The KEY declaration defines character positions 1 to 9 of each database record as an INFOS II key. Because the first nine character positions are the Last Name field of the database records, the last names in each record become INFOS II keys.

Figure 8-17 shows what EMPLOYEE and the database linked to it look like after you execute the command file. Now you can find an employee's database record either by knowing his/her last name or his/her employee number.

The prior examples worked with one INFOS II index file. The next example works with two: VENDORS and PARTS, shown in Figure 8-18.

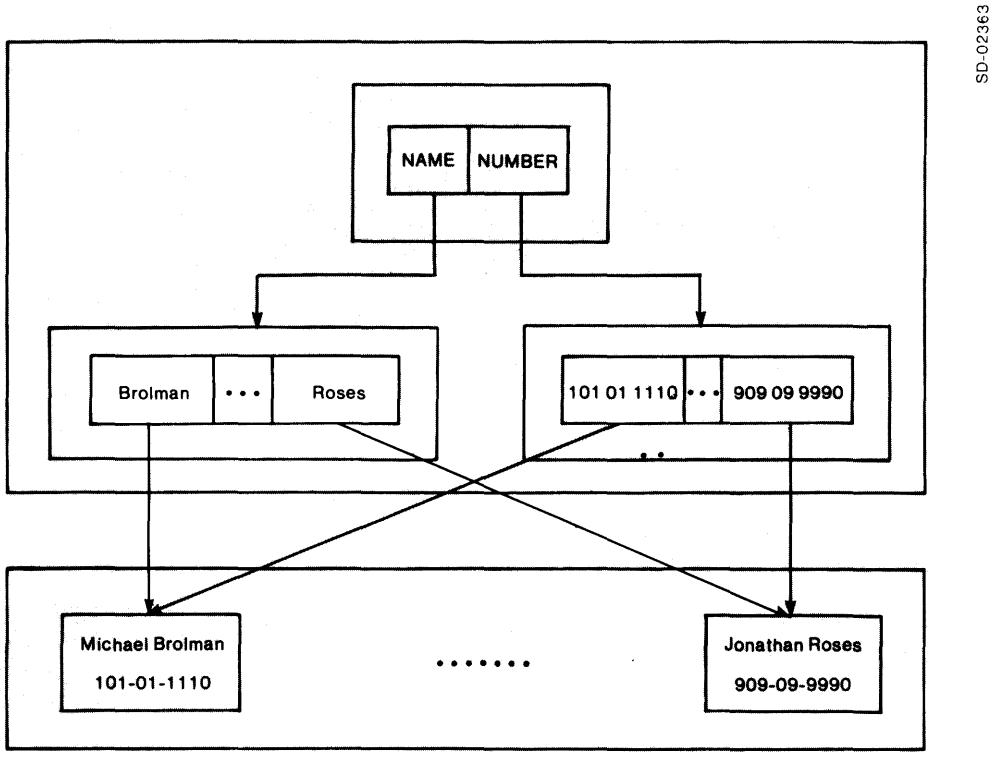


Figure 8-17. EMPLOYEE, after Command File Executes

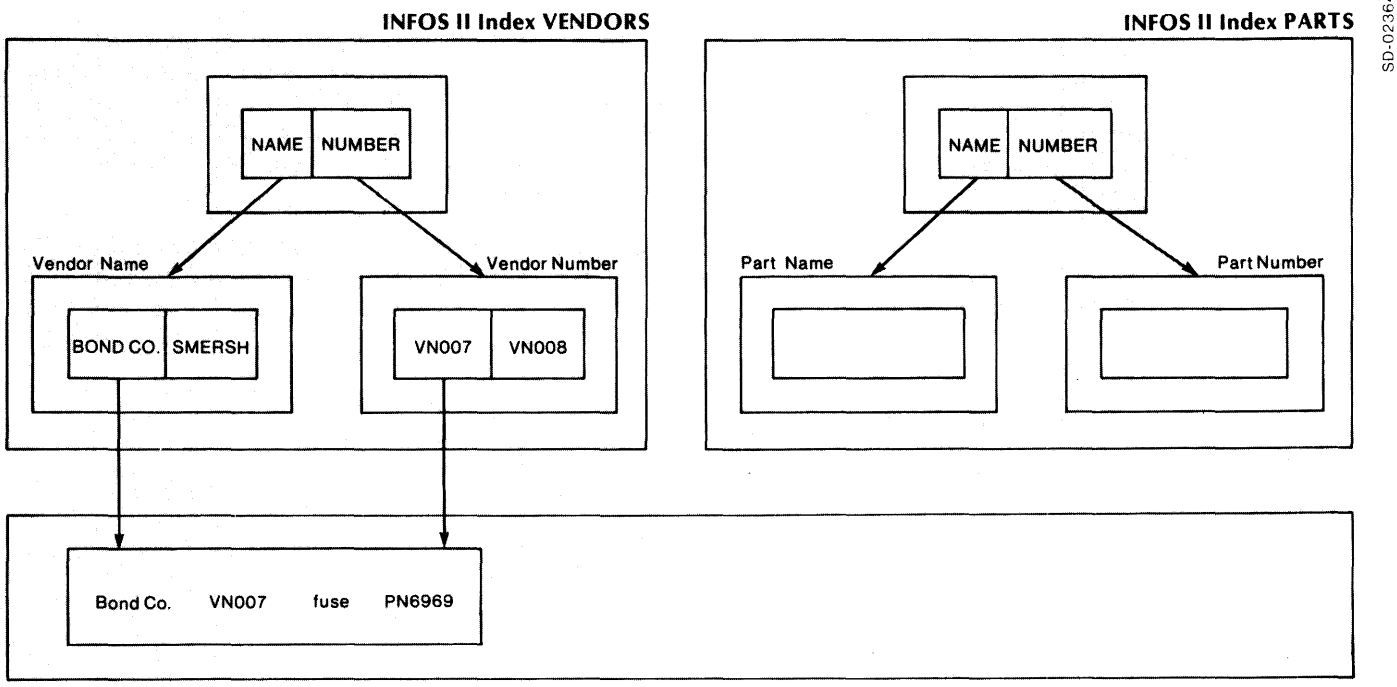


Figure 8-18. VENDORS and PARTS, before First Command File Executes

The main subindex of VENDORS has two keys: NAME and NUMBER. NAME refers to a vendor's name and NUMBER refers to a vendor's number. The subindex linked to NAME is the NAME subindex; the subindex linked to NUMBER is the NUMBER subindex. NAME and NUMBER subindexes' keys are linked to the same database records. Figure 8-19 shows the database records' format.

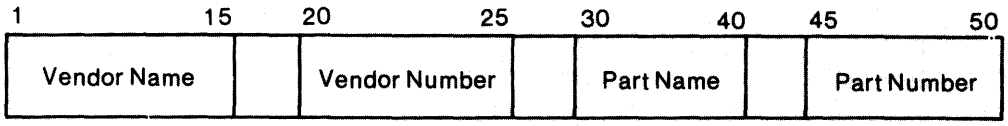


Figure 8-19. Format of Records Shared by VENDORS and PARTS

PARTS's main subindex also has two keys: NAME and NUMBER. In this subindex, however, NAME refers to a part's name and NUMBER to a part's number. Though the subindexes linked to NAME and NUMBER exist, they are empty.

You want to make each database record's Part Name field an INFOS II key for the NAME subindex of PARTS. Also, you want to make each database record's Part Number field an INFOS II key for the NUMBER subindex of PARTS. You'll need a separate command file to accomplish this second task. In both cases, you'll link all these keys to the same database records to which VENDORS is linked.

Here is the command file which makes the Part Name field into keys, and links them to the database:

```
INPUT INFOS INDEX IS "VENDORS",
        PATH IS "NAME",
                *, RECORD,
        RECORDS ARE 50 CHARACTERS.
OUTPUT INFOS INVERSION IS "PARTS",
        PATH IS "NAME",
                *.
KEY 30/40.
COPY.
END.
```

The INPUT INFOS declaration lets you gain access to the database records through the NAME subindex keys. You could just as easily gain access to these records through the NUMBER subindex. In the OUTPUT INFOS declaration, you specify the INVERSION option because you'll link keys in PARTS's NAME subindex to the same database records to which VENDORS is linked. The PATH IS clause tells the utility to put the INFOS II keys in the NAME subindex. The KEY declaration defines which range of characters in the database records will become the INFOS II keys. Because character positions 30 to 40 are the Part Name field, this field becomes the INFOS II key.

Figure 8-20 shows how the NAME subindex of PARTS is linked to the database after you execute the command file.

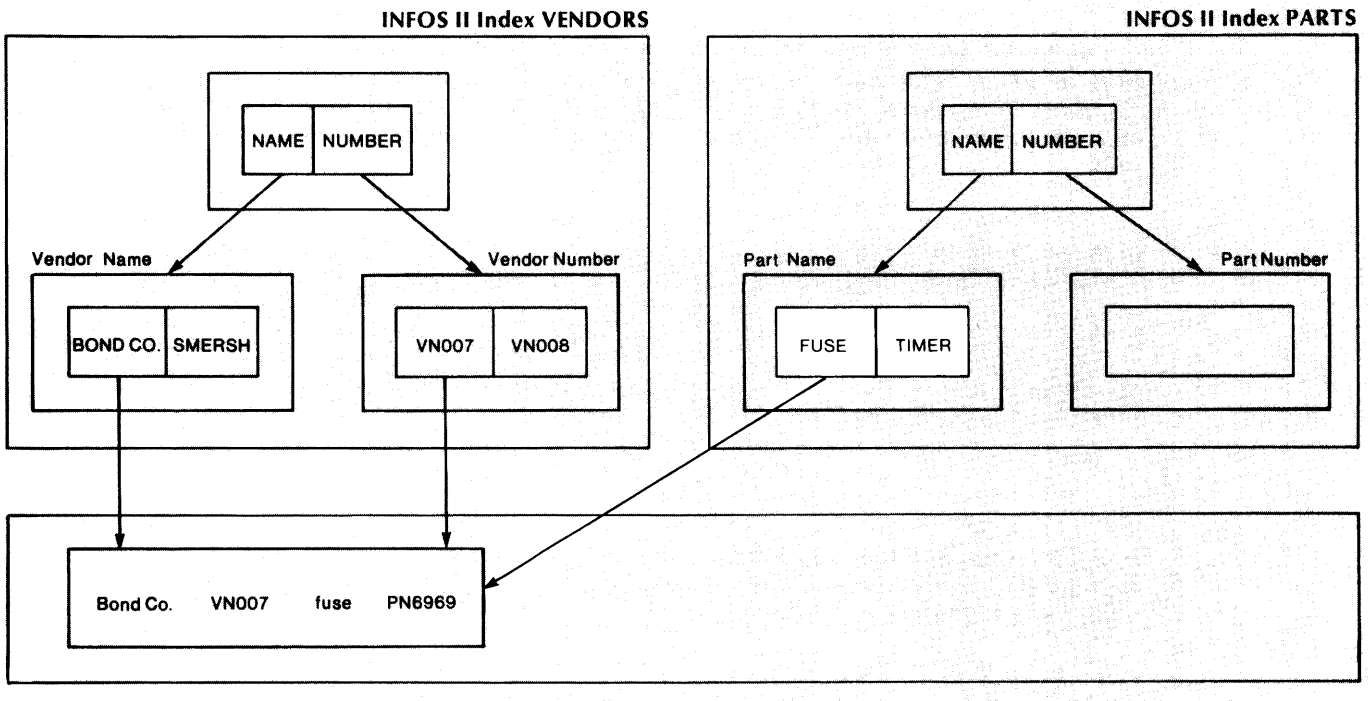


Figure 8-20. VENDORS and PARTS, after First Command File Executes

Here is the command file which makes the Part Number field into keys, and links them to the database:

```
INPUT INFOS INDEX IS "VENDORS",
        PATH IS "NAME",
                *, RECORD,
        RECORDS ARE 50 CHARACTERS.
OUTPUT INFOS INVERSION IS "PARTS",
        PATH IS "NUMBER",
                *.
KEY 45/50.
COPY.
END.
```

This command file is very similar to the previous one. The INPUT INFOS declaration is the same. The OUTPUT INFOS declaration specifies the INVERSION option because the INFOS II keys will be linked to the same database to which VENDORS's keys are linked. The PATH IS clause tells the utility to put the INFOS II keys in the NUMBER subindex of PARTS. The KEY declaration defines the range of characters in the database records which will become the INFOS II keys. Because character positions 45 to 50 are the Part Number field, this field becomes the INFOS II key.

Figure 8-21 shows you how the NAME and NUMBER subindexes are linked to the database after you execute both command files.

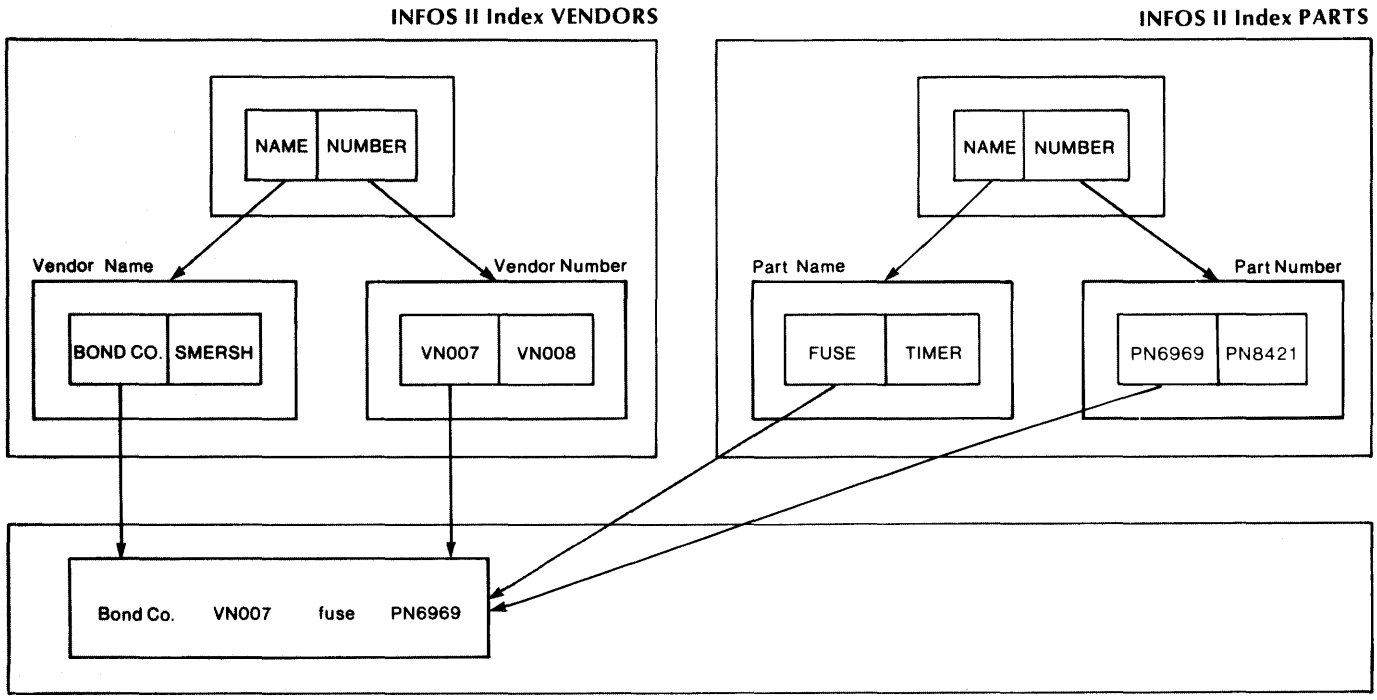


Figure 8-21. VENDORS and PARTS, after Second Command File Executes

End of Chapter

Chapter 9

Report Writer

This chapter describes

- the interface between Sort/Merge and Report Writer
- the .QFORMS and .RFORMS files that Report Writer uses
- RWCHECK, a Stand-Alone Compiler

If you know the Report Writer of INFOS II QUERY with Report Writer, you already know the Report Writer of Sort/Merge. They both work the same way.

Report Writer takes data and puts it into a report. Figure 9-1 shows a sample report made from INFOS II database records. Two files control how Report Writer makes a report: .QFORMS and .RFORMS. The .QFORMS file contains a *qformat* and the .RFORMS file contains an *rformat*. Basically, a qformat defines fields in records; an rformat defines a report's layout by selecting how and where to use those fields.

CAUCUS CAR PARTS

Dollar Value of Warehouse Inventory 12/09/80

Description	Part Number	Stock	Cost	Dollar Value	
CM	056793	0	39.50	$ 0.00	
			Value of CM	Stock =	$ 0.00
BRA PD	057211	3	13.12	$39.36	
	057217	5	13.26	$66.30	
	057219	7	13.21	$92.47	
			Value of BRA PD	Stock =	$198.13
HTR BX	061523	1	36.00	$36.00	
			Value of HTR BX	Stock =	$36.00
BRA LN	062251	4	4.11	$16.44	
			Value of BRA LN	Stock =	$16.44
WHL CY	066234	2	10.11	$20.22	
			Value of WHL CY	Stock =	$20.22
				Total Value of Warehouse Stock	$270.79

Figure 9-1. Sample Report

Interface between Sort/Merge and Report Writer

The Sort/Merge OUTPUT REPORT declaration lets you interface with Report Writer. The OUTPUT REPORT declaration's format is

```
OUTPUT REPORT IS "name1",

[BLOCKS ARE integer CHARS,]

[ELEMENTS ARE integer BLOCKS,]

RFORMAT IS "name2" FROM FILE "name3"

[, QFORMS FILE IS "name4"] .
```

name1	is the output file. It accepts records from either INFOS II or AOS files, and it contains the report.
name2	is the name of the rformat contained in the .RFORMS file. The rformat is a group of statements which governs report layout.
name3	is the name of the .RFORMS file, without the .RFORMS extension.
name4	is the name of the .QFORMS file. You don't have to include the clause containing name4 if the .RFORMS and the .QFORMS files have the same name before the . (period).

BLOCKS ARE Clause

The BLOCKS ARE clause states the number of characters (bytes) in a physical block of a magnetic tape file. Use this clause if the file that you're writing to doesn't have the default AOS blocksize of 2,048 characters.

Specifying a large block size reduces I/O time and saves tape.

ELEMENTS ARE Clause

The ELEMENTS ARE clause describes the file's size in physical units on the disk. The default element size is one (512 bytes). See "Increase Element Size" in Appendix E for an explanation of why you would want to specify the element size.

The .QFORMS File

Report Writer uses qformats, contained in a .QFORMS file, to define the structure of records. In this section, we give you the information you need to build a .QFORMS file.

Building the .QFORMS File

You store the data definitions for your file as text in the < name >.QFORMS file, where < name > is name4 in the OUTPUT REPORT declaration format. For example, CAUCUS.QFORMS contains the data definitions for the file CAUCUS. You can build this text file from the CLI with a text editor.

Syntax of the .QFORMS File

The .QFORMS file consists of one or more qformats. You may define many different qformats in the .QFORMS file. These qformats can represent unique record types or different views of the same record type. In either case, each qformat includes a list of field descriptions that contain the following information:

TITLE	A name for the field, up to 30 characters in length.
START BYTE	The byte on which the field starts.
END BYTE	The byte on which the field ends.
TYPE	The data type within the field (an optional entry).

NOTE: The system assumes that the field is in ASCII characters unless you specify otherwise.

A qformat consists of three types of lines: one *START_FORMAT* line, one or more *field descriptor* lines, and one *END_FORMAT* line.

Although you do not have to position these lines in specific columns, you must use TAB to delimit the fields, and you must use NEW LINE (↓) to delimit lines. We represent the TAB character as <tab>.

START_FORMAT Line

The first line of each qformat is the START_FORMAT line. This line contains two fields: the keyword START_FORMAT, followed by a tab, and the qformat name that contains up to 32 characters, including blanks, (without tabs) and followed by a NEW LINE. For example:

```
START_FORMAT<tab>THIS IS A NAME↓
```

Field Descriptor Lines

Each qformat contains from one to 30 field descriptor lines after the START_FORMAT line. The field descriptor lines contain either three or four fields, depending on whether you include the "type" field for non-ASCII formatting.

The first field is a title of up to 30 characters (not including a tab), followed by one or more tabs. The second field contains the start-byte integer, followed by a tab. The third field contains the end-byte integer followed by either a tab if you include an optional data type, or a NEW LINE if you do not include an optional data type.

Both start- and end-byte fields may contain leading blanks or tabs so that you can set up the integers in columns.

The fourth, optional field is the data type field containing one of the following data types:

DECIMAL TSO
DECIMAL LSO
DECIMAL LSS
DECIMAL TSS
DECIMAL
BIN INTEGER
BIN FLOAT
PACKED
ASCII

Where

TSO is Trailing Sign Overpunch.
LSO is Leading Sign Overpunch.
LSS is Leading Sign Separate.
TSS is Trailing Sign Separate.

The data type you specify must be followed by a NEW LINE.

Remember, if you do not specify an optional data type (see "Non-ASCII Formatting," below), then the field will be ASCII by default. For example:

FIRST.....FIELD < tab > 1 < tab > 15 ↵

SECOND FIELD < tab.tab > 22 < tab > 28 < tab > ASCII ↵

LAST ONE < tab....tab > 51 < tab > 56 < tab > DECIMAL LSS ↵

Even though only the second field is specifically designated, both the first and second fields are ASCII data types. The last field is a DECIMAL LSS data type.

END_FORMAT Line

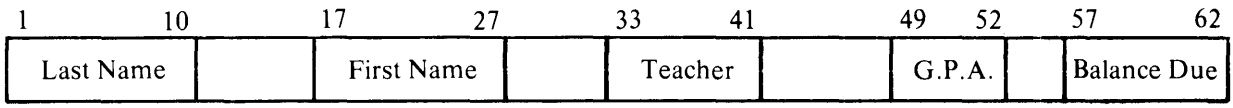
After specifying all the desired field descriptor lines, you must include a last line that says END_FORMAT, followed by a NEW LINE (↵). For example:

```
END_FORMAT↵
```

Example

Let's look at a student record format and a qformat which defines and labels fields in the records that the format represents.

The records' format is



Qformat RECORD is

```
START_FORMAT    RECORD
LAST NAME       1          10
FIRST NAME      17         27
TEACHER         33         41
GPA             49         52
BALANCE DUE     57         62
END_FORMAT
```

In this qformat, the first field descriptor line defines bytes 1 through 10 as a field, and labels it LAST NAME. Thus, the start-byte of LAST NAME is 1, and its end-byte is 10. Qformat RECORD similarly defines and labels the other fields: FIRST NAME, TEACHER, GPA, and BALANCE DUE. We'll use qformat RECORD again in the summary example which concludes this chapter.

Setting Up a .QFORMS File

Selection Procedures

Keep in mind that you may arrange field descriptor lines in any order. This is useful when you want fields printed in a different order than they appear in the original record.

Non-ASCII Formatting

It is easiest to use qformats when all the data records are ASCII values. Because ASCII is the default display mode, you name only the different byte ranges that you need to access.

Many files, however, especially those created and maintained by COBOL programs, contain non-ASCII data. If you were to display these records directly on a display terminal without changing the data type, they would be visually incomprehensible. By using the appropriate field description types, Report Writer converts the values into readable form (numbers).

To use this Report Writer feature for converting non-ASCII data, you will have to know how many bytes are in the internal representation of each data type. Then, you can correctly position the start- and end-bytes of the respective fields.

Size and Scope of the .QFORMS File

Your .QFORMS file may contain up to ten separate qformats. If you need more space for qformats (without deleting any you have already defined), you can create another .QFORMS file using the CLI.

Error Messages

RWCHECK, a Stand-Alone Compiler, checks qformat syntax for errors. If it detects an error, RWCHECK sends one of the messages listed in Appendix C. We'll discuss RWCHECK after we explain how to construct a .RFORMS file.

The .RFORMS File

Report Writer takes INFOS II database or AOS records and rearranges them into a formatted report, along with whatever headings, page breaks, and computations you want the file to contain.

An rformat, contained in a .RFORMS file, defines the eventual report that Report Writer produces. In this section, we describe the creation of rformats in detail, and give you the reference material you'll need to build the .RFORMS file.

As you read this section, consider how Report Writer uses the rformat. When invoked, Report Writer reads through the rformat, making note of each different descriptor line and the action each requires. It then reads one record at a time and performs the actions the rformat has requested.

You should understand qformats before reading this section. If you are not familiar with the qformats designed for your system, or with the .QFORMS file and how to build it, see the section, "The .QFORMS File."

Building the .RFORMS File

You create rformats in a CLI text file called < name > .RFORMS, where < name > is name3 in the OUTPUT REPORT declaration format. You can build this text file from the CLI with a text editor.

Rformat Definition

An rformat consists of a number of different *format descriptor lines*. Each line has a limited number of fields, separated by tabs. As we describe each line, you'll see how flexible rformat definition is. You must follow certain simple rules when creating rformats; however, the resulting reports can be as complex or simple as you want.

Rformats must begin with a START_REPORT line and end with an END_REPORT line. As we shall see, between these first and last lines, the other types of format descriptor lines may appear. Blank lines may appear anywhere within the rformat. In fact, you should feel free to use blank lines often as a means of grouping statements.

As we explain the individual format descriptor lines, remember that these lines tell Report Writer what actions to take, the types of information to display, and the size and shape of the report.

The command line description and terminology used in defining rformats are

UPPERCASE LETTERS	Keywords
lowercase letters	Variables
< tab >	TAB: delimiter between arguments
)	NEW LINE: end-of-line delimiter
data_item	A data field described in the qformat file
def_item	A data field described by a DEFINE line

We will introduce some basic rules about creating rformats here, and then explain them in further detail later on.

Although there are various Report Writer format descriptor lines that you must include in the rformats, the order of these lines is highly flexible. Just follow the basic rules, and Report Writer will check for syntax errors.

- The rformat always begins with a START_REPORT command and ends with an END_REPORT command.
- The QFORMAT command line must precede any command line that makes reference to a data_item.
- You must define a field in a DEFINE line before using it in any other line.
- You can define a given def_item only once per rformat.
- Any item (def or data) can only have one PICTURE per rformat.
- COL/LIN and LIN/PAGE command lines each appear only once in an rformat.
- A DETAIL command line for an item must precede any BREAK command line calling for underlining that item.

START_REPORT Line

The first line of each rformat is the START_REPORT line. It signals the beginning of an rformat by naming it.

This line contains two fields: the keyword START_REPORT, followed by a tab, and the name of the report which contains up to 32 characters followed by a NEW LINE. For example:

START_REPORT <tab> report_name↵

Sort/Merge cannot handle rformat names that have blanks (for example, Caucus Car Parts). If an rformat name contains blanks, abbreviate name2 (in the OUTPUT REPORT declaration) by eliminating the blanks.

Keeping the rformat name unique is a good idea. Let's say you have one rformat called STOCK_VALUE and one called STOCK_INVENTORY. If you try to refer to either rformat by STOCK, the Report Writer will interpret it as the first one defined in the .RFORMS file. Therefore, either use one word names for rformats or make sure the first word of the name is unique.

Comment Line

If you want to make notes when building an rformat, use the comment line. The comment line gives no instructions to Report Writer; it's for documentation only.

The comment line consists of an exclamation point (!) in the first column followed by your comment. You can include any number of comment lines in a rformat. For instance, the person at Caucus Car Parts who built the STOCK VALUE rformat included the following comments:

```
!
! This produces a report with the total value of the current
! warehouse stock
!
```

These comments will not show up on the report produced by this rformat. However, anyone who looks at the .RFORMS file will be able to read them.

QFORMAT Line

The QFORMAT command line specifies which qformat in the .QFORMS file you want the Report Writer to use when it accesses the records. Each qformat defines different data items. The QFORMAT line allows you to refer to the various data items by the names used in one of these qformats.

This line consists of two fields: the keyword QFORMAT, followed by a tab, and the qformat name, followed by a NEW LINE.

For example:

QFORMAT <tab> QNAME↵

Lines Per Page

You can specify how many lines per page you wish the report to contain. The default is 60 lines per page; the minimum allowed is 30. Remember that your system's line printers may have different top and bottom margins. For instance, if a line printer is automatically set to leave a three line margin at top of the page, given a 66 line page, only 63 lines are available for printing. Check with your system manager when and if you need to adjust these margins, or see The AOS Command Line Interpreter User's Manual (093-000122) for more details about the Forms Control Utility.

The line consists of two fields: the keyword LIN/PG, followed by a tab, and the number you wish to specify, followed by a NEW LINE. The LIN/PG line takes the following form:

LIN/PG <tab> n↵

where n is the number of lines per page.

Columns Per Line

You can specify the width of the printed report page, in number of characters. The default is 80 characters.

This line also has two fields: the keyword COL/LIN, followed by a tab, and the number you wish to specify, followed by a NEW LINE. The COL/LIN line takes the following form:

COL/LIN <tab> n↵

HEADER Lines

You use HEADER lines to define report and column headers. HEADER lines tell Report Writer to display header information at the top of each page of the report. In other words, if the printed report runs more than one page, the header information appears on each page.

You can define up to ten lines of header information in an rformat. You don't have to define them in any particular order, or all together. Report Writer will check that items on the same HEADER line do not overlap.

The HEADER line consists of four fields: the keyword HEADER, the line number on which the header information is to appear, the location on the line where the header information is to appear, and what header information is to print there. A tab follows the first three fields, a NEW LINE follows the last field. A HEADER line would look like the following:

HEADER <tab> line # <tab> loc <tab> item↵

line # is number of the HEADER line 1 through 10

loc is
- C centered
- n start at byte n
- -n start n bytes from the end

item is
- "literal"
- 'literal'
- data_item
- DATE (today's date)
- PAGE (the current page number)

The PAGE field is nine characters long. The DATE field is eight characters long.

You can have more than one HEADER line for the same line number; Report Writer combines these specifications to determine the final line. For instance, if you want to define five column heads on the same line, your HEADER lines might look like the following:

```
HEADER 6   5    "Description"
HEADER 6   20   "Part Number"
HEADER 6   35   "Stock"
HEADER 6   45   "Cost"
HEADER 6   60   "Dollar Value"
```

Although you have used five HEADER lines, you have told the Report Writer to actually display only one line of HEADER information.

These five HEADER lines do not need to appear in this order or even contiguously. In fact, in the rformat called STOCK_VALUE, these lines are scattered throughout the file.

If you have more than one line of header information in your report, you do not have to specify all the line numbers. If you skip a line number, Report Writer will interpret it as a blank line in your printed report. For instance, if you have three lines of header information defined to appear on lines 1, 3, and 5, Report Writer will interpret lines 2 and 4 as blanks. After all the lines of header information have

been displayed, Report Writer skips one more line before displaying the detail information. You do not have to specify a blank line between the header information and the detail information; Report Writer automatically does it for you.

If you specify a data_item as part of your header information, the value of that specified item will be the same as its value on the first detail line for that page. In other words, Report Writer finds that item's value from the same record it uses for the first line of detail information in the report.

DEFINE Lines

You use DEFINE lines to define temporary variables which are considered part of the record for the duration of the Report Writer process. In effect, you're asking Report Writer to temporarily extend the record to include new items. The Report Writer performs calculations to derive these temporary items.

This line consists of three fields: the keyword DEFINE, followed by a tab, a name for the defined item, a tab, a simple arithmetic expression, and a NEW LINE. The DEFINE line takes the following form:

DEFINE <tab> def_item <tab> expression↵

An expression involves exactly one operator (+, -, /, *, %) and any two of the following: data_items, def_items, and numeric constants.

The temporary variable defined in the DEFINE line is called a def_item. The results of the calculations resulting from a DEFINE line are not retained from record to record; they are calculated anew for every record. You may define up to ten temporary variables.

For instance, if an input record has the fields, PART_NBR, QUANTITY, and PRICE, you might use the following DEFINE lines:

```
DEFINE COST     PRICE * QUANTITY
DEFINE PROFIT   2 % COST
```

In this case, you can ask Report Writer to display any of the six items, including COST and PROFIT, at some point in the report.

DETAIL Lines

You use DETAIL lines to specify which items to print from each record. A DETAIL line tells Report Writer what to include as detail information. You may define up to 5 lines of DETAIL information per report. As with the HEADER line, you may specify more than 5 DETAIL lines in order to create 5 lines of detail information. Report Writer checks for overlapping fields on every detail line.

The DETAIL line consists of four fields: the keyword DETAIL, the number of the DETAIL line (1 through 5), where on the page the detail information will appear, and the item you are specifying. The first three fields are followed by a tab; the item itself is followed by a NEW LINE. The DETAIL line takes the following form:

DETAIL <tab> line # <tab> loc <tab> item↵

line # is number of the DETAIL line 1 through 5

loc is C centered
        n start at byte n
        -n start n bytes from the end

item is "literal"
         'literal'
         data_item
         def_item

In the rformat called STOCK_VALUE, five different items from the same record were needed for the report. A different DETAIL line is used for each item, even though all five items will appear on the same line in the report. The same number, 1, appears in each DETAIL line, indicating it is detail line 1. This rformat could have defined up to four more lines of detail information.

PICTURE Lines

You use the PICTURE line to specify the way you want a particular item to appear in the printed report. If you do not define a PICTURE line, Report Writer outputs any numeric data in a 14 character field with 10 signficant digits and two digits to the right of the decimal. By numeric data we mean all data_items described as numbers in the qformat, and all def_items.

The PICTURE line consists of three fields: the keyword PICTURE, followed by a tab, an item name followed by a tab, and the picture clause, followed by NEW LINE. The PICTURE line takes the following form:

PICTURE <tab> item <tab> pic

item is a data_item
a def_item

pic is a picture clause

PICTURE allows you to specify a field format using a subset of the PL/1 and COBOL picture facilites. Report Writer recognizes seven symbols, as explained in Table 9-1.

A $, if present, must always precede 9s and Vs. Only one V is legal per picture. An S may only be at either end of a picture, while floating Ss are always at the beginning. Floating signs (S) imply no legal $s and floating dollar signs ($) imply one S at most. Commas (,), periods (.), and slashes (/) are simply cosmetic characters and may go anywhere in the picture.

For example, here are five different picture clauses for the signed number -43.20:

SSSS9.V99 would result in -43.20
S999.V99 would result in - 43.20
99V.99S would result in 43.20-
S99V99 would result in -4320
S9999 would result in - 43

Table 9-1. Picture Characters

Picture Character	Symbol Definition	Usage
9	Numeric digit	Indicates a numeric digit, 0 through 9.
$	Dollar sign	Indicates a currency symbol or a numeric digit.
V	Floating decimal point	Indicates decimal point location.
S	Signed number	Indicates a sign or a digit.
/	Slash	Indicates a slash character.
.	Period	Indicates a period character.
,	Comma	Indicates a comma character:

BREAK Lines

A BREAK line tells Report Writer to suspend all other actions long enough to take a specified action. There are several actions which you may optionally specify

- print a line of summary information.
- suppress repeated fields in DETAIL lines (NO_REP).
- skip to a new page, or to a new line (POST_BREAK_SPACE, PRE_BREAK_SPACE, PAGE_EJECT).
- underline specific columns (UNDERLINE).

Report Writer performs the break action when it encounters a change in the value of a specified data item. Any summary information printed by a break action can take only one physical line on the report. However, you can specify BREAK actions on up to five different data items. In other words, you can tell Report Writer to take a break action when up to five data values change.

A BREAK line can take one of two different syntactic forms: in one form it tells Report Writer to print summary information at the time of the break action; the other form tells Report Writer about optional printing or spacing features.

In either case, the first two fields consist of the keyword BREAK, followed by a tab, and the name of the data item whose changing value will cause the break action to occur, also followed by a tab.

The BREAK line takes either of the following forms:

BREAK <tab> data_item <tab> loc <tab> item↵

or

BREAK <tab> data_item <tab> print_op↵

loc is
- C centered
- n start at byte n
- -n start n bytes from the end

item is
- "literal"
- 'literal'
- data_item
- COUNT
- TOTAL (data_item)
- AVG (data_item)
- MIN (data_item)
- MAX (data_item)
- def_item
- TOTAL (def_item)
- AVG (def_item)
- MIN (def_item)
- MAX (def_item)

print_op is
- NO_REP
- POST_BREAK_SPACE
- PRE_BREAK_SPACE
- PAGE_EJECT
- UNDERLINE (data_item)
- UNDERLINE (def_item)

The keyword PRE_BREAK_SPACE tells Report Writer that you want a blank line *before* any summary information. The keyword POST_BREAK_SPACE directs Report Writer to put a blank line *after* printing summary information. PAGE_EJECT tells Report Writer to eject a page after all break actions have been taken. And, UNDERLINE tells Report Writer to underline the specified item.

Note that there is not a tab between the keyword and its argument (for instance UNDERLINE (data_item)).

The data_item specified in a BREAK line does not have to appear in a DETAIL line. Note, however, to make sense, the records should be sorted somehow on this key. In other words, if the records are not naturally sorted by the specifed data_item, any computed subtotals would be meaningless.

If multiple break actions occur after the same DETAIL line, the lines of break information are printed in the order in which they appear in the rformat. All break actions are triggered after the last detail and before any total lines.

In addition to printing data_items, def_items, and constants, however, Report Writer also allows you to print summary items (total, average, minimum, maximum, and count) of data_items and def_items as break information. The summary item calculations are done by Report Writer; you can not maintain these calculations. Report Writer will check that no fields overlap on any line of break information.

If the BREAK line describes optional printing or spacing features, a keyword will generally suffice to tell Report Writer what is desired. However, UNDERLINE requires a reference to a DETAIL line field as well, so Report Writer knows what to underline.

TOTAL Lines

TOTAL lines tell Report Writer to print summary information at the end of the report. Total information can occupy up to 10 lines on the report and can include calculations performed by Report Writer.

TOTAL lines consist of four fields: the keyword TOTAL, the TOTAL line number, the locations at which the information appears, and the information you want in the TOTAL line. The first three fields are each followed by a tab; the last field is followed by a NEW LINE.

The TOTAL line takes the following form:

TOTAL <tab> line # <tab> loc <tab> item

line # is	the TOTAL line number
loc is	C centered n start at byte n -n start n bytes from the end
item is	"literal" 'literal' data_item COUNT TOTAL (data_item) AVG (data_item) MIN (data_item) MAX (data_item) def_item TOTAL (def_item) AVG (def_item) MIN (def_item) MAX (def_item)

Total information can include data_items, def_items, summary information for these items, and constants. Report Writer does any necessary calculations required by summary items, and checks that data fields don't overlap on the report's total information lines.

Although total information appears on the report only once, at the end of the report, you can use multiple TOTAL lines if the total information includes multiple fields. Total information can occupy up to 10 lines on the report.

END_REPORT Line

The last line of an rformat is always an END_REPORT line. It signals the end of the rformat.

This line consists of one field, the keyword, END_REPORT, followed by a NEW LINE.

For example:

```
END_REPORT)
```

Users of INFOS II QUERY with Report Writer Please Note

Sort/Merge's report writer neither supports the INFOS II QUERY with Report Writer SORT lines nor accepts them in an rformat. Sort your data by using Sort/Merge command file statements.

Designing Reports

The various format descriptor lines we've just described make up an rformat. Some of these lines may appear once per rformat, others may appear many times, and some you may not need at all. In fact, you can define an rformat with as little as a START_REPORT line, an END_REPORT line, and one DETAIL line, or just one BREAK line or just one TOTAL line. As long as you have the starting line and the ending line, and at least one of these other lines, you'll have a legal rformat.

As we've said before, you use a text editor to define an rformat. Once you've entered the various format descriptor lines, making sure to follow the simple rules associated with each, you can verify the rformat's syntax using RWCHECK, the Stand-Alone Compiler.

Before using your newly designed rformat to produce a report, it's a good idea to produce a sample page. In addition to verifying the syntax of any rformat you've created, RWCHECK can also provide a sample report. You can see what the report will look like; if you're not satisfied with the results, you simply edit the rformat. You can repeat these steps as many times as necessary to produce a satisfactory report format.

Size and Scope of the .RFORMS File

Your .RFORMS file may contain up to ten different rformats. If you try to define more than ten rformats in one .RFORMS file, the Report Writer won't recognize them. If you need more space for your rformats, (without deleting any you've already defined), you can create another .RFORMS file using the CLI.

RWCHECK - The Stand-Alone Compiler

RWCHECK is an interactive utility that verifies the syntax of qformats and rformats. It also provides a sample page of your report.

You can use RWCHECK in batch mode, as well as interactively, depending on the switch used when invoking it. You invoke RWCHECK from the CLI, using the following syntax:

RWCHECK [*/R=rformat name* | */S=filename* | */E=filename* | */L=listfile* | */Q=name.QFORMS* | */A* | */N*] *< name > [.RFORMS]*

< name > .RFORMS	is the name of the file containing defined rformats.
/R=rformat name	allows you to specify the particular *rformat name* within the .RFORMS file you want verified.
/S=filename	allows you to designate the *filename* to which you want a sample page sent.
/E=filename	sends any errors and the lines on which these errors occur to *filename*.
/L=listfile	sends the entire rformat and any errors to a named *listfile*.
/Q=name.QFORMS	allows you to specify a .QFORMS file that is not *< name > .QFORMS*.
/A	specifies that all rformats in the named .RFORMS file are to be validated.
/N	specifies no interaction. No questions will appear on the screen during the RWCHECK session.

Keep in mind the following rules when you use the switches:

- You can not use a /A switch with a /R switch.
- The /N switch also requires that you use a /R or a /A switch and that you specify the .RFORMS file as an argument.
- If you do not use the /L switch, all errors are sent to @ OUTPUT.
- The /S switch tells the RWCHECK to create a sample page and send it to the named file. If you don't use this switch, you must indicate whether you want a sample page and where you want it sent during the RWCHECK session.

When you invoke the RWCHECK, an interactive dialog begins (except when you use the /N switch). During this dialog, RWCHECK asks various questions and displays the default answer in brackets ([ ]).

The only argument used with the RWCHECK command is the .RFORMS filename. If you do not include the argument, RWCHECK asks you for the name of a file. For example:

```
) X RWCHECK↵
RFORMS Filename:
```

When you type the .RFORMS filename, RWCHECK session continues

```
RFORMS Filename: CAUCUS↵
Would you like to see a list of report format names? [N] Y↵
STOCK_VALUE
OLDIES
Report Format Name: STOCK_VALUE↵
```

If there are any errors, a message appears telling you on which line the error exists. For example, let's say you defined the following rformat, called BOGUS.

```
START_REPORT  BOGUS
BREAK         PRE_BREAK_SPACE
END_REPORT
```

On first glance it looks right. You've got the starting and ending lines and at least one BREAK command line. However, when you verify it with RWCHECK, the following information appears on your terminal:

```
START_REPORT BOGUS
BREAK          PRE_BREAK_SPACE
*** Wrong number of arguments for this command: BREAK LINE: 2
END_REPORT
*** There MUST be a detail, total, or break line somewhere.
*** TOTAL ERRORS:      2
```

RWCHECK indicates an error in the BREAK line. In addition, since this is the only format descriptor line in the rformat (other than the starting and ending lines) and it's erroneous, it doesn't register as a legal format descriptor line. Therefore, you receive the second error message.

If you've defined an error-free rformat, RWCHECK responds with

No errors detected. Want a sample page? [Y]

(Note that you get this question only if you omitted the /S switch when invoking the RWCHECK.)

If you answer Y, the RWCHECK then asks

Name a file ...any file: [@ OUTPUT]

You can then indicate the name of a text file to which you want the sample page sent, or you can take the default and let the RWCHECK send the sample page to @ OUTPUT.

Whether you want a sample page or not, RWCHECK next asks if you want to validate another rformat:

Do you wish to validate another report format? [N]

If you do, RWCHECK repeats the entire dialog. If you don't, the session ends, and you're back in the CLI.

Summary Example

The Prolman Institute for Gifted Children (a small private school) wants a report of its tenth grade class. We show you what's needed to produce this report. We start first with the unsorted file of student records, and then explain the command file, qformat, and rformat you need. After that, we show a sample report page produced by RWCHECK, and end with the finished report.

Each student record states a student's name, teacher, Grade Point Average (G.P.A), and balance of tuition owed. The format of the 62-character fixed-length records is

1 10		17 27		33 41		49 52		57 62
Last Name		First Name		Teacher		G.P.A.		Balance Due

Input file GRADE_10, which contains these records, is

BOURKE	JARON	KATZ	3.87	000.00
GREEN	PAMELA	BERGERON	3.33	250.00
JERNSTEDT	KAREN	MCCARTHY	3.65	175.00
CLARK	ROBERT	MCCARTHY	3.29	750.00
HUMPHREYS	PATRICIA	BERGERON	3.10	500.00
BOURKE	ANDREW	KATZ	3.90	000.00
MARTIN	LARRY	BERGERON	3.75	600.00
NORTON	PAMELA	MCCARTHY	4.00	000.00
HARRIS	LINDA	KATZ	3.75	375.00
LOPEZ	ANNA-MARIE	BERGERON	3.00	500.00
ROSE	SUSAN	BERGERON	3.50	250.00

From these records, the Institute wants to create a report containing

- records sorted first by teacher and second by G.P.A.
- titles over each column
- average G.P.A for each class
- average student G.P.A.
- total balance due

The following command file tells the utility how to sort the records and directs it to the appropriate qformat and rformat.

```
INPUT FILE IS "GRADE_10", RECORDS ARE 63 CHARACTERS.
OUTPUT REPORT IS "G10_REP",
    RFORMAT IS "FORMATTER" FROM FILE "G10".
KEY 33/41.
KEY 49/52 DESCENDING.
SORT.
END.
```

Let's analyze this command file. The KEY declarations make the Teacher field (character positions 33 through 41) the primary key, and make the G.P.A. field (character positions 49 through 52) the secondary key. The OUTPUT REPORT declaration tells the utility to send the report to output file G10_REP, and to use rformat FORMATTER, contained in .RFORMS file G10. Because we also named the .QFORMS file G10, the utility by default uses the qformat RECORD of G10.QFORMS.

We saw qformat RECORD earlier in this chapter when we discussed qformats. Again, qformat RECORD is

```
START_FORMAT    RECORD
LAST NAME       1       10
FIRST NAME      17      27
TEACHER         33      41
GPA             49      52
BALANCE DUE     57      62
END_FORMAT
```

This qformat's field descriptor lines amount to a statement of the character positions of each field in the student records.

Rformat FORMATTER, which uses qformat RECORD, is

```
START_REPORT    FORMATTER
QFORMAT         RECORD

HEADER  4           C       "Prolman Institute for Gifted Children"
HEADER  6           C       "Grade Ten Report"

HEADER  9           1       "Teacher"
HEADER  10          1       "********"
DETAIL  1           1       TEACHER
BREAK   TEACHER             PRE_BREAK_SPACE
BREAK   TEACHER             POST_BREAK_SPACE
BREAK   TEACHER             POST_BREAK_SPACE
BREAK   TEACHER             3    "CLASS AVERAGE ="
BREAK   TEACHER 36          AVG (GPA)

HEADER  9           19      "Last Name"
HEADER  10          19      "**********"
DETAIL  1           19      LAST NAME

HEADER  9           30      "First Name"
HEADER  10          30      "***********"
DETAIL  1           30      FIRST NAME

HEADER  9           45      "G.P.A."
HEADER  10          45      "******"
DETAIL  1           46      GPA

HEADER  9           55      "Balance Due"
HEADER  10          55      "************"
DETAIL  1           57      BALANCE DUE

TOTAL   1           1       "Average Student G.P.A. = "
TOTAL   1           36      AVG (GPA)

TOTAL   2           1       "Total Balance Due = "
TOTAL   2           49      TOTAL (BALANCE DUE)

END_REPORT
```

Now let's analyze this rformat. Notice that we grouped lines of the rformat together and separated these groups by a blank line. This makes the rformat easier to read. It also lets us refer to the rformat by groups of lines, for example, the first group, the next group, etc. You might want to skip over the following explanation of the groups and procede to compare FORMATTER directly with the sample page produced by RWCHECK (Figure 9-2) and with the final report (Figure 9-3).

The first group names the rformat in the START_REPORT line and names the qformat in the QFORMAT line.

The next group tells Report Writer to center the report's title ("Prolman Institute ... Report").

The teacher group performs the greatest number of functions of all the groups. First, it starts the column title "Teacher" in column 1 of the line printer page, 9 lines down. Second, it underscores "Teacher" with asterisks (*). Third, it places the teachers (already sorted by Sort/Merge) in the "Teacher" column. Report Writer knows where to find the teachers because the data_item TEACHER in the DETAIL line was defined in the qformat RECORD. Compare where Report Writer places the teachers in the report (column numbers 1 through 7) with where the teachers appear in input file GRADE_10 (character positions 33 through 41). Remember that you can take a field from a record and put it anywhere you want in the report. And fourth, this group's BREAK lines tell Report Writer that you want some action taken before the teacher changes in the "Teacher" column. For example, one teacher change is from Bergeron to Katz. The action is to print "Class Average = " followed by the average for one teacher's class. Report Writer calculates the class average for you by using AVG, one of many functions listed earlier in the section "BREAK Lines." The BREAK line which contains the PRE_BREAK_SPACE print option tells Report Writer to skip a line before it prints a class average. The two BREAK lines which contain the POST_BREAK_SPACE print option tell Report Writer to skip two lines after it prints a class average.

The next four groups place the "Last Name," "First Name," "G.P.A.," and "Balance Due" column titles on the same line number as "Teacher." Then they underscore the titles with asterisks (*) and place the appropriate field under each title.

The next to last and last groups tell Report Writer to print summary information about all the records. The next to last group tells Report Writer to print "Average Student G.P.A." and then print the average. Notice that we use the same AVG function which the last BREAK line of the Teacher group uses. We get the average of all student G.P.A.s instead of the average of only all students in one class, because this time AVG appears in a TOTAL line. The last group tells Report Writer to print "Total Balance Due" and then the total.

The END_REPORT line signals the end of FORMATTER.

Before you execute the command file (and after you've corrected any errors spotted by RWCHECK), you can use RWCHECK to get an approximate idea of what the final report will look like. The RWCHECK command line:

```
RWCHECK/S=SAMPLE↵
```

sends the sample page shown in Figure 9-2 to file SAMPLE.

Prolman Institute for Gifted Children

Grade Ten Report

Teacher	Last Name	First Name	G.P.A.	Balance Due
*******	**** *****	**********	******	***********
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXXX	XXXX	XXXXXX
Class Average =			9999.99	
Average Student G.P.A. =			9999.99	
Total Balance Due =				9999.99

Figure 9-2. Sample Page Produced by RWCHECK

Finally, after you execute the Sort/Merge command file, you'll find the report shown in Figure 9-3 in output file G10_REP.

Prolman Institute for Gifted Children

Grade Ten Report

Teacher	Last Name	First Name	G.P.A	Balance Due
BERGERON	MARTIN	LARRY	3.75	600.00
BERGERON	ROSE	SUSAN	3.50	250.00
BERGERON	GREEN	PAMELA	3.33	250.00
BERGERON	HUMPHREYS	PATRICIA	3.10	500.00
BERGERON	LOPEZ	ANNA-MARIE	3.00	500.00
Class Average =			3.33	
KATZ	BOURKE	ANDREW	3.90	000.00
KATZ	BOURKE	JARON	3.87	000.00
KATZ	HARRIS	LINDA	3.75	375.00
Class Average =			3.84	
MCCARTHY	NORTON	PAMELA	4.00	000.00
MCCARTHY	JERNSTEDT	KAREN	3.65	175.00
MCCARTHY	CLARK	ROBERT	3.29	750.00
Class Average =			3.64	
Average Student G.P.A. =			3.55	
Total Balance Due =				3400.00

Figure 9-3. Prolman Institute Report

End of Chapter

Appendix A

Command Line and Command File Statement Summary

This appendix summarizes the utility's command lines and command file statements. We order statements by type:

- declarations
- imperatives
- massage statements
- END statement

Command Line

```
{SORT  }  {/C  [/T=filename]}  [{/L          }]  [/N]  [/O]  [/S]  [ INTO outfile [FROM infile ... ]]
{MERGE }  {/c=filename      }  [{/L=filename }]
```

Command File Statements

Declarations

INPUT FILE

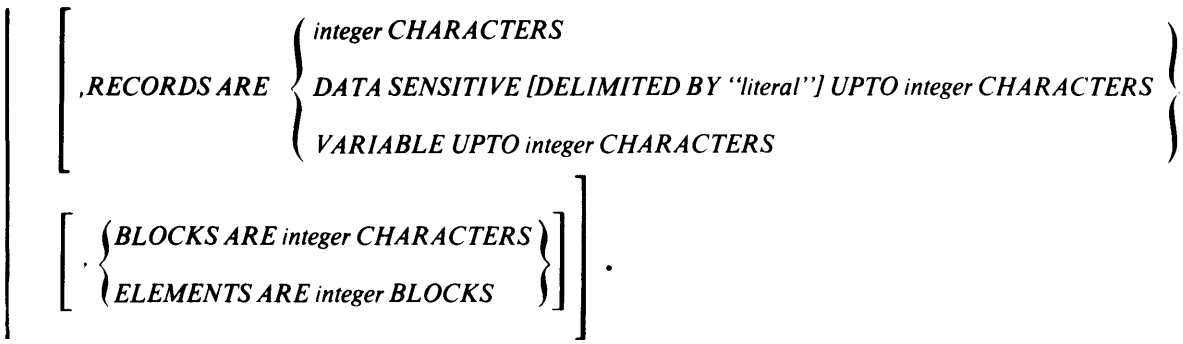
INPUT FILE IS "name"

```
[ ,RECORDS ARE { integer CHARACTERS                                                    }
               { DATA SENSITIVE [DELIMITED BY "literal"] UPTO integer CHARACTERS       }
               { VARIABLE UPTO integer CHARACTERS                                      }

  [, BLOCKS ARE integer CHARACTERS ] ]  .
```

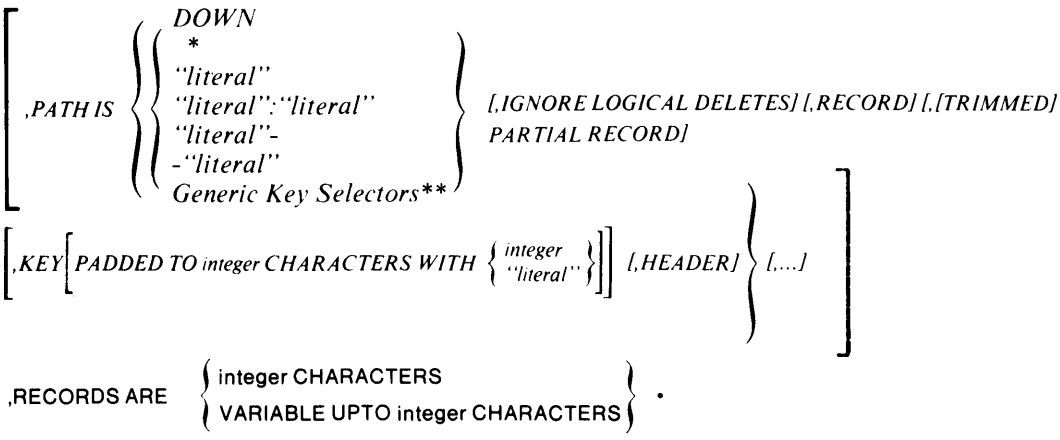
OUTPUT FILE

OUTPUT FILE IS "name"

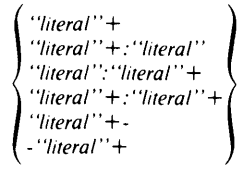


INPUT INFOS

INPUT INFOS INDEX IS "name"



**Generic Key Selectors*



OUTPUT INFOS

OUTPUT INFOS { INDEX / INVERSION } IS "name"

[[, *RECORD IS* { *integer/integer* / *integer/LAST* }] [, *PARTIAL RECORD IS* { *integer/integer* / *integer/LAST* }]

[, *TRIM* [{ *integer* / *"literal"* } *FROM*] *KEYS*]

[, *PATH IS* [{ *down* / *"literal"* / *"literal"*+ } , [*...*,]] *]] .

OUTPUT REPORT

OUTPUT REPORT IS "name1",

[BLOCKS ARE integer CHARS,]

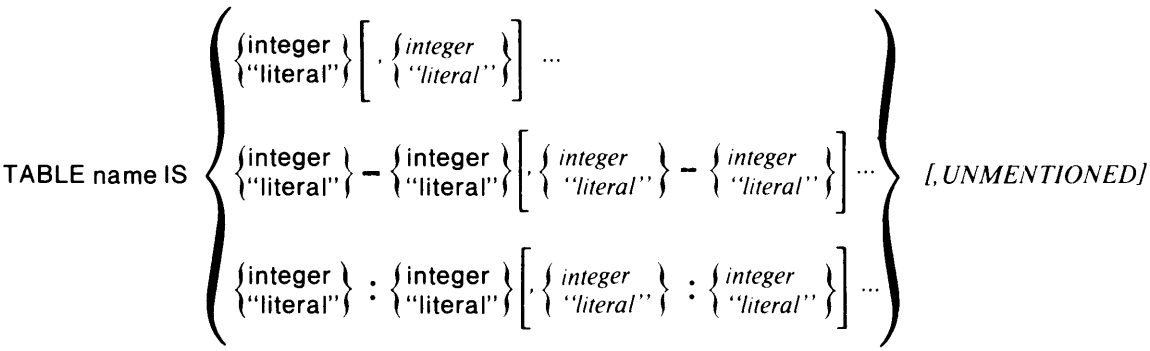
[ELEMENTS ARE integer BLOCKS,]

RFORMAT IS "name2" FROM FILE "name3"

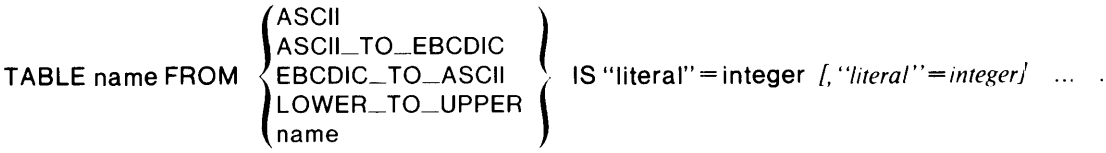
[, QFORMS FILE IS "name4"] .

TABLE

FORMAT ONE



FORMAT TWO



FORMAT THREE

TABLE name1 IS FILE "name2".

KEY

FORMAT ONE

```
      { integer/integer      }                              [ { ASCENDING  } ]
KEY   { integer/LAST         }  [COLLATED BY tablename]     [ { DESCENDING } ]
      { integer1:integer2    }
```

FORMAT TWO

```
                                           [ { LOP } ]
                            { DECIMAL      [ { TOP } ] }
                            {              [ { LSS } ] }
      { integer/integer   } [ TYPE IS      [ { TSS } ] ] [ { ASCENDING  } ]
KEY   { integer1:integer2 } [ { PACKED               } ] [ { DESCENDING } ]
                            [ { BINARY               } ]
                            [ { FLOAT                } ]
                            [ { EXTERNAL FLOAT       } ]
```

WORK FILE

WORK FILE IS "filename".

Massage Statements

COMPRESS

```
                                   { LEFT  [ { integer   } FILLED ] }
COMPRESS   { integer/integer }     {       [ { "literal" }        ] }
           { integer/LAST    }     { RIGHT [ { integer   } FILLED ] }   USING tablename .
                                   {       [ { "literal" }        ] }
                                   { VARIABLE                       }
```

IF

IF condition [{ AND / OR } *condition*] ... THEN { STOP / SKIP *["filename"]* / REFORMAT massage statement / REPLACE massage statement / REPLACE TABS massage statement / INSERT massage statement / PAD massage statement / TRANSLATE massage statement / COMPRESS massage statement } .

The format of the *condition* phrase is:

{ { "literal" / integer/integer / integer/LAST } { = / < / > / <= / =< / >= / => / <> / :=: / :<>: } { "literal" / integer/integer / integer/LAST }

RECORDCOUNT { = / < / > / <= / =< / => / >= / <> } integer }

INSERT

INSERT { "literal" / RECORDCOUNT / TAG } { BEFORE integer / IN { integer/integer / integer/LAST } / AFTER LAST }

PAD

PAD TO integer CHARACTERS WITH { "literal" / integer } .

REFORMAT

REFORMAT { integer/integer / integer/LAST } , { integer/integer / integer/LAST } *[,...]* .

REPLACE

REPLACE [{ *ALL* / *ANY* }] "literal_1" IN { integer/integer / integer/LAST } WITH "literal_2" .

REPLACE TABS

REPLACE TABS IN { integer/integer / integer/LAST } WITH { "literal" / integer } *[, TAB STOPS ARE integer [, integer] ...]*.

TRANSLATE

```
TRANSLATE  {integer/integer}  USING  {ASCII_TO_EBCDIC  }
           {integer/LAST    }        {EBCDIC_TO_ASCII  }
                                     {LOWER_TO_UPPER   }
                                     {tablename        }  .
```

Imperatives

[STABLE] [TAG] SORT [DELETING DUPLICATES].

MERGE [DELETING DUPLICATES].

COPY.

END Statement

END.

End of Appendix

Appendix B

Error Messages

This appendix lists error messages you can receive if you make a mistake. There are two major classes of error messages: semantic and abort. We'll start with the semantic class.

Semantic Error Messages

Semantic error messages have the following format:

Semantic error message
EXECUTION INHIBITED - integer ERROR(S) WAS(WERE) DETECTED

This part of the message	tells you
semantic error message	the error. These errors are listed separately in Table B-1. If you use the /L or /L=filename switch in the command line, the message appears (in the list file) under the statement containing the error. (We explain these switches in Chapter 7)
integer	number of errors detected

Table B-1 lists alphabetically most of the semantic error messages you can receive. In the "Cause" column of Table B-1, we sometimes tell you to inspect a statement. We do this if the message is almost self-explanatory. Taken together, the message and the statement to inspect will lead you to find your error.

Table B-1. Semantic Error Messages

Message	Cause
ALL REPLACE WILL NEVER TERMINATE	The string you want replaced is part (or all) of the string which will replace it
ATTEMPT TO REFERENCE UNDEFINED TABLE	Inspect TRANSLATE and/or COMPRESS statement
CANNOT DEFAULT FILE DESCRIPTION FOR FILES NOT OF FIXED OR DATA SENSITIVE TYPE	The records in the input file(s) aren't of the fixed-length or data-sensitive type
CANNOT REWRITE A PERMANENT FILE	You tried to overwrite an output file that has the permanent attribute
CHARACTER *char* WAS PREVIOUSLY MENTIONED IN THIS TABLE	You tried to define the collating or translate value of *char* twice in the same TABLE declaration
COULD NOT RENAME TO THAT FILENAME	You cannot use the file you named in response to: FILE NAME TO USE FOR COMMAND FILE: The utility will repeat this question

(continues)

Table B-1. Semantic Error Messages

Message	Cause
DATA SENSITIVE FORMAT NOT ALLOWED FOR INFOS FILES	You included a RECORDS ARE DATA SENSITIVE clause in an INPUT INFOS declaration
DOWN MAY NOT BE QUALIFIED	You paired an extractor phrase with the key selector DOWN
ERROR IN ACCESSING OUTPUT FILE	There's a problem with one or more files in pathname to the output file: • file(s) do not exist • you don't have proper access to file(s) because of a wrong directory or searchlist, or wrong Access Control List. For example, you need Write access to the output file.
ERROR IN ACCESSING WORK FILE	There's a problem with pathname to work file. See ERROR IN ACCESSING OUTPUT FILE
FIELD OUT OF RANGE	A location phrase specifies character positions beyond the limits of a record. For example, you specify 1/555 when the longest record is 100 characters.
FILE IS A DIRECTORY	The last file in a pathname is a directory
FILE NAME TO USE FOR COMMAND FILE:	The utility's response if you answer yes to the question: SAVE INPUT FOR EDITING AND RESUBMISSION? (Y or N)
FIRST ELEMENT IN A : OR - GROUP MUST BE LESS THAN SECOND	Inspect TABLE declaration
"FROM" TABLE NAME IS UNDECLARED	You didn't declare a table name in the FROM phrase of a TABLE declaration. For example, in the declaration: `TABLE EX FROM ERROR IS "A"=1.` ERROR is not declared.
HEADER MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one HEADER (an extractor) with one key selector. For example: `*, HEADER, HEADER`
IF YOU DO NOT SPECIFY SORT OR MERGE YOU MUST PROVIDE A COMMAND FILE	Inspect the command line

(continued)

Table B-1. Semantic Error Messages

Message	Cause
IMPROPER COMMAND LINE SYNTAX	In the command line, you either • omitted INTO or FROM, or • didn't place INTO or FROM correctly
INFOS KEYS ARE LIMITED TO 255 CHARACTERS	Inspect KEY declaration used to define INFOS keys
INFOS KEYS MUST BE OF CHARACTER TYPE	The command file has a KEY declaration containing a TYPE IS clause, and an OUTPUT INFOS declaration. For example: OUTPUT INFOS INDEX IS "OOPS". KEY 1/LAST TYPE IS PACKED
INFOS OUTPUT ALLOWS ONLY A SINGLE KEY	Your command file contains an INFOS output file and more than one KEY declaration
INSUFFICIENT ACCESS TO OUTPUT FILE'S PARENT DIRECTORY	You need Write or Append access to the output file's parent directory
INSUFFICIENT MEMORY TO START THE SPECIFIED MERGE	One or more of these is true: • there are too many skip files • your records and set of tape buffers are too large • TAG SORT requires more memory
INTEGER MUST BE IN RANGE 0 TO 255	The ASCII decimal equivalent of a single-character literal must be from 0 through 255, inclusive
INVALID KEY SPECIFICATION FOR OUTPUT FILE	You used a non-unique key selector in an OUTPUT INFOS declaration. For example: * (above the lowest subindex level), -"Z", "A"-, or "A"-"Z".
INVALID RANGE SPECIFICATION	In a location phrase, the integer before the slash is greater than the integer after the slash. For example: 9/5.
INVALID TYPE FOR WORK FILE	Work files must be on disk
INVERSION REQUIRES ALL INPUT FILES TO BE INFOS FILES	

(continued)

Table B-1. Semantic Error Messages

Message	Cause
KEY MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one KEY (an extractor) with one key selector. For example: *, KEY, KEY
LAST ELEMENT IN OUTPUT PATH MUST BE AN *	The last key selector in the PATH IS clause of the OUTPUT INFOS declaration must be an *
LITERAL MUST BE A SINGLE CHARACTER	
MAXIMUM NUMBER OF USER TAB STOPS (20) EXCEEDED	Inspect REPLACE TABS massage statement
MERGE REQUIRES TWO OR MORE INPUT FILES	
MISSING COMMAND FILE	You invoked Sort/Merge in noninteractive mode (without /C) and didn't name a command file in the command line
MULTIPLE OUTPUT FILES	You can have only one output file • in a command file • in a command line not used with a command file • between a command line and a command file For example, if your command line is SORT/C=DO_IT INTO FILE_OUT FROM FILE_IN you cannot declare an output file in the command file DO_IT
MULTIPLE PARTIAL RECORD RANGE DEFINITION	You included more than one PARTIAL RECORD IS clause in the OUTPUT INFOS declaration
MULTIPLE RECORD RANGE SPECIFICATIONS	You included more than one RECORD IS clause in an OUTPUT INFOS declaration
MULTIPLE TRIMMED KEY SPECIFICATIONS	You included more than one TRIM KEYS phrase in the OUTPUT INFOS declaration
MUST DEFINE INPUT FILE NAME	Sort/Merge received no input files from either the command line or the command file
NO DEFAULT FILENAME FOR /T SWITCH	You didn't set /T equal to a filename

(continued)

Table B-1. Semantic Error Messages

Message	Cause
NON CHARACTER KEYS MUST HAVE CONSTANT RANGE BOUNDS	You used LAST in the location phrase of a KEY declaration and the keys are not of the character type. For example: KEY 1/LAST TYPE IS DECIMAL.
/O IS REQUIRED TO ALLOW OVER-WRITE OF AN EXISTING FILE	The output file exists before you execute the command file, and you didn't include /O in the command line
ONLY CHARACTER KEYS MAY SPECIFY AN ALTERNATE COLLATING SEQUENCE	You used a COLLATED BY phrase and a TYPE IS phrase in the same KEY declaration
OUTPUT FILE UNSPECIFIED	Sort/Merge did not receive an output file from either the command line or the command file; there was neither an output file declaration in the command file, nor INTO outfile in the command line
PADDED KEY MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one KEY PADDED TO (an extractor) with one key selector. For example: *, KEY PADDED TO 3 CHARS WITH "-", KEY PADDED TO 5 CHARS WITH "+"
PARTIAL RECORD MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one PARTIAL RECORD (an extractor) with one key selector. For example: *, PARTIAL RECORD, PARTIAL RECORD
PARTIAL RECORDS ARE LIMITED TO 255 BYTES	The range specified in the location phrase of an OUTPUT INFOS declaration's PARTIAL RECORD IS clause exceeds 255 bytes
PREMATURE END OF INPUT	You either • forgot the END statement in the command file, or • typed CTRL-D to escape entering the command file from the console
READ ACCESS REQUIRED FOR INPUT FILES	Inspect the Access Control List for each input file
RECORD MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one RECORD (an extractor) with one key selector. For example: *, RECORD, RECORD
RECORD SPECIFICATION IS INCOMPATIBLE WITH INVERSION	You can't use the RECORD IS clause in an OUTPUT INFOS declaration when you invert keys

(continued)

Table B-1. Semantic Error Messages

Message	Cause
REPORT FORMAT WAS NOT VALIDATED	Report format checker did not approve format. A message from the checker accompanies this message
SAVE INPUT FOR EDITING AND RESUBMISSION? (Y OR N)	You receive this message if you're entering the command file at the console, and the statements you typed would be lost. You're most likely to receive this message if you type NEW LINE before correcting an error
SKIP FILE ALREADY EXISTS	The skip file must not exist before you execute the command file
SYNTAX ERROR: [[LINE *INTEGER*]]	The *integer* is the line number containing the error. You receive the optional [LINE *integer*] when you use the /L or /L=filename switch in the command line. (This is explained in Chapter 7.) Hint: sometimes a trivial spelling error can raise this error
TABLE FILE COULD NOT BE OPENED	The table declared in a TABLE declaration either • doesn't exist, or • is not accessible because of a wrong directory or searchlist, or a wrong Access Control List
TABLE FILE MUST CONTAIN AT LEAST 255 BYTES	The file named in the TABLE declaration must contain at least 255 bytes
TABLE NAME MULTIPLY DEFINED	You named the same file in more than one TABLE declaration
TAG SORT DOES NOT ALLOW KEY OVERLAPS	There's more than one KEY declaration of the form KEY integer/LAST And, in the same command file, TAG SORT is the imperative.
TAG SORT REQUIRES NON-INFOS DISK FILES	You tried to tag sort with an INFOS input, tape, or character device
THE EXPRESSION IS TOO COMPLEX	There are too many operators (=, <, >, etc.) in an IF massage statement

(continues)

Table B-1. Semantic Error Messages

Message	Cause
TRIMMED PARTIAL RECORD MAY ONLY APPEAR ONCE PER LEVEL	You paired more than one TRIMMED PARTIAL RECORD (an extractor) with one key selector. For example: *, TRIMMED PARTIAL RECORD, TRIMMED PARTIAL RECORD
UNABLE TO ACCESS INPUT FILE	There's a problem with one or more files in the pathname to the input file: • file(s) do not exist • you don't have proper access to file(s) because of a wrong directory or searchlist, or wrong Access Control List
UNABLE TO CREATE SCRATCH FILE	There's a problem with the pathname to scratch file. See the previous error message. Thus, the utility could not ?CREATE a default work file
UNABLE TO OPEN THE COMMAND FILE	There''s a problem in the pathname to command file. See UNABLE TO ACCESS INPUT FILE
UNABLE TO RESOLVE INPUT PATHNAME	There's a problem in the pathname to input file. See UNABLE TO ACCESS INPUT FILE
UNABLE TO RESOLVE OUTPUT FILE PATHNAME	There's a problem in the pathname to output file. See UNABLE TO ACCESS INPUT FILE
'UNMENTIONED' MAY ONLY APPEAR ONCE	You used UNMENTIONED in a TABLE declaration more than once
WORK FILES REQUIRE BOTH READ AND WRITE ACCESS	
WRITE ACCESS TO OUTPUT FILE IS REQUIRED	

(concluded)

Abort Error Messages

There are different subclasses of abort error messages:

- I/O failure
- skip file
- key comparisons and massaging
- execution phase
- initialization phase
- other

I/O Failure Error Messages

All I/O failure error messages have a pathname in them. This feature distinguishes them from the other subclasses of abort error messages.

The format of the I/O failure error message is:

ABORT
Phase identifier
Sort error message
Pathname
AOS or INFOS II error

This part of the message	tells you
Phase identifier	which phase of the utility found the error. The phases are ● replacement selection ● intermediate merge ● final-user merge ● copy-filter
Sort error message	which error the utility found. These errors are listed separately in table B-2
Pathname	the pathname of file causing the error
AOS or INFOS II error message	which error AOS or INFOS II found. For an explanation of the AOS or INFOS II errors, see the *AOS Programmer' Manual* or the *INFOS II System User's Manual (AOS)*, respectively

Table B-2. Sort Error Messages for I/O Failure

Message	Comment
?CLOSE OF RECORD I/O FILE	In general, record I/O files are
?OPEN OF RECORD I/O FILE	• labeled magnetic tape
?READ FROM RECORD I/O FILE	• generic files (@OUTPUT)
?WRITE TO RECORD I/O FILE	• queue files (@LPT)
	• character devices (@CRA)
CLOSE OF INFOS FILE	Messages with INFOS FILE in them may occur while processing an INFOS file
INFOS RETRIEVE SUBINDEX DEFINITION CALL	
OPEN OF INFOS FILE	
READ FROM INFOS FILE	
RETRIEVE KEY FROM INFOS FILE	
WHILE TRAVERSING SUBINDEXES OF INFOS FILE	
WRITE TO INFOS FILE	
FLUSHING PARTIAL BUFFER TO BLOCK I/O FILE	Block I/O files are disk and unlabeled tape files
?GCLOSE OF BLOCK I/O FILE	
?GCLOSE OF WORK FILE	
?GOPEN OF BLOCK I/O FILE	
?GOPEN OF WORK FILE	
?RDB FROM BLOCK I/O FILE	
READING RECORD FROM BLOCK I/O BUFFER	
?WRB TO BLOCK I/O FILE	
WRITING RECORD INTO BLOCK I/O BUFFER	

Skip File Error Messages

The format of the skip file error message is

ABORT
Phase identifier
Sort error message
AOS error message

This part of the message	tells you
Phase identifier	which phase of the utility found the error. The phases are ● replacement selection ● final-user merge ● filter-copy
Sort error message	which error the utility found. These messages are ?CLOSE OF SKIP FILE ?OPEN OF SKIP FILE ?WRITE TO SKIP FILE
AOS error message	which error AOS found. For an explanation of these errors, see the *AOS Programmer's Manual.*

Key Comparison and Massaging Error Messages

The format of key comparison or massaging error messages is

ABORT
Phase identifier
Sort error message
Record locator

This part of the message	tells you
Phase identifier	which phase of the utility found the error. The phases are ● replacement selection ● intermediate merge ● final-user merge ● copy-filter
Sort error message	which error the utility found. These errors are listed separately in Table B-3
Record locator	at or near which input record the error occurred. The record locator has its own format: ERROR OCCURRED AT OR NEAR INPUT RECORD nnn where nnn is an integer. For key comparison messages, nnn is always approximate.

Table B-3. Sort Error Messages for Key Comparison and Massaging Errors

Message	Cause
COMMERCIAL OR FLOATING POINT TRAP IN KEY COMPARE	One of these is true: • a DECIMAL or PACKED field contained an invalid character. • an EXTERNAL FLOAT number could not be represented in internal form • a FLOAT number was not in correct format
INSERT BEFORE LOCATION NOT IN RECORD	A record is not the correct length for the requested massaging action
INVALID CHARACTER IN EXTERNAL FLOAT FIELD	
OVERFLOW IN EXTERNAL FLOAT FIELD CONVERSION	EXTERNAL FLOAT numbers are converted to internal numbers in order to be compared
RANGE SPECIFICATION TOO LONG FOR RECORD	A record is shorter than it is specified in the massage statement
RECORD LONGER THAN PAD LENGTH	
RECORD TOO SHORT FOR KEY	A field specified in a KEY declaration is not in the record

Execution Phase Error Messages

The format of the execution phase error message is

ABORT
Phase identifier
CANNOT START EXECUTION OF PHASE

[INSUFFICIENT MEMORY AVAILABLE
NOT ENOUGH CHANNELS FOR ALL FILES
BUFFER REQUIREMENTS FOR FILE(S) EXCESSIVE]

The phase identifier tells you which phase of the utility found the error. The phases are

- replacement selection
- intermediate merge
- final-user merge
- copy-filter

Usually, attempts to execute very complex command files cause these errors. Here are two actions you can take to reduce command file complexity:

- use fewer skip files

- restructure the command file into two smaller files; accomplish in two executions what you tried to accomplish in one

Initialization Phase Error Messages

The format of the initialization phase error message is

ABORT
FROM INITIALIZATION PHASE:
Sort error message
[Message]

This part of the message	tells you
Sort error message	which error the utility found. These errors are listed separately below
Message	about another error. It could be an AOS error message, for example. This message is optional

Some of the sort error messages for initialization phase errors are

ABANDONING COMMAND FILE SCAN DUE TO:

[UNRECOVERABLE SYNTAX ERROR
PARSE STACK OVERFLOW - STATEMENT TOO COMPLEX
INPUT BUFFER OVERFLOW: CHECK FOR UNTERMINATED LITERAL
UNTERMINATED LITERAL AT END OF FILE
SYMBOL TABLE OVERFLOW: USE FEWER TABLES]

?CREATE OF SKIP FILE FAILED

I/O ERROR ON HELP FILE

UNABLE TO OPEN /L FILE

UNABLE TO OPEN @OUTPUT

Other Error Messages

You very rarely raise the error messages in this section. They usually indicate

- tampering with directories and files the utility uses while sorting, or
- improper installation of the product

The format of these error messages is

ABORT
Phase identifier
Sort error message
[AOS error message]

This part of the message	tells you
Phase identifier	which phase of the utility found the error. The phases are ● replacement selection ● intermediate merge ● final-user merge ● copy-filter
Sort error message	which error the utility found. These errors are listed separately below.
AOS error message	which error AOS found. For an explanation of the AOS error, see the *AOS Programmer's Manual*

The sort error messages for these errors are

?CHAIN TO NEXT PHASE FAILED
?CLOSE OF DIRECTORY
?CREATE OF OUTPUT FILE
?DELETE OF OLD VERSION OF OUTPUT FILE
ERROR IN I/O TO A SORT INTERNAL DATA FILE
?FSTAT BEFORE RE-CREATION OF OUTPUT FILE
?GNFN ERROR
?GOPEN OF INPUT FILE FOR TAG RECORD RECOVERY
?GTMES FOR /PID SWITCH
INEXPLICABLE ?GNAME ERROR
INEXPLICABLE ?PSTAT ERROR
INEXPLICABLE ?RUNTM ERROR
INVALID /PID SWITCH ARGUMENT
?OPEN OF DIRECTORY
?PROC OF RWCHECK FAILED
?RENAME OF TEMP OUTPUT FILE TO FINAL NAME

The following message should not occur:

SORT INTERNAL LOGIC ERROR

If it does occur, call you local systems engineer.

End of Appendix

Appendix C

RWCHECK Error Messages

This Appendix includes explanations of the error messages you might receive while using Report Writer. We also suggest ways to correct these errors.

The error messages are arranged as follows:

- Qformat syntax
- Report Writer runtime
- Rformat syntax

Qformat Syntax Errors

RWCHECK checks the syntax of all qformats. If there is a syntax error in a qformat, you'll receive any of the following messages.

Error Message	Cause
Bad END BYTE on line --x--	Query doesn't like the ending byte you specified on the named line. Be sure to use a valid number (a positive integer) greater than or equal to the start byte.
Bad KEY TYPE on line --x--	You specified an invalid key type on the named link.
Bad START BYTE on line --x--	Query doesn't like the starting byte you specified on the named line. Be sure to use a valid number (a positive integer).
Looking for a START_FORMAT on line --x--	The Query can't find a valid START_FORMAT line. Check your syntax.
START BYTE is greater than END byte on line --x--	The first byte on the named line is larger than the last byte.
START_FORMAT must have a qformat name - on line --x--	There's an error in the START_FORMAT line. Have you specified a valid qformat name?
Too many Field Descriptor Lines in qformat.	You've used too many field descriptor lines in the qformat. The maximum allowed is 30.
Too many qformats in .QFORMS file	You've defined too many qformats for one .QFORMS file. One .QFORMS file can hold only 10 qformats.
--x-- is not a valid format	This message follows all other syntax error messages as a reminder that the named qformat is not valid.

Report Writer Runtime Errors

When the utility detects a runtime error, it inserts one of the following error messages into the Report Writer's output file. The utility also inserts:

THERE WERE n RUNTIME ERRORS DURING REPORT GENERATION

into the statistical output if any runtime errors were generated.

Error Message	Cause
CIS fault	You defined a field in your qformat as a non-ASCII data type. The data the Report Writer found in that field is not consistent with the non-ASCII data type. Check your data and qformat.
Divide by zero error. Zero result assigned:	Your DEFINE line included a division operation and on this input record the divisor for the DEFINE is a zero. Division by zero is undefined. The Report Writer uses this message to let you know that this has happened and that it set the def_item to zero.
Invalid numeric value:	You've used a Picture clause for a non-numeric field or else there is non-numeric data in what should be a numeric field. Check your data.
Significant digits truncated:	Your number won't fit in the defined or default Picture clause. The Report Writer truncated the extra digits. Check your picture clause.
The input record is not long enough	The Report Writer is trying to find data in fields which would be beyond the end of the record. Check your qformat and data.

Rformat Syntax Errors

You'll get rformat syntax errors when you run RWCHECK or when you attempt to activate the rformat. The errors must be fixed before Report Writer will let you activate the rformat.

General Rformat Syntax Errors

You'll receive on of the following messages if a syntax error occurs in any of the format descriptor lines that make up an rformat.

Error Message	Cause
Ambiguous RFORMAT command: -----	The Report Writer doesn't recognize the command; the abbreviation is not unique.
Command line item is too large, line ignored	You specified an argument longer than 50 characters. Only comments may have items which are longer than 50 characters.
Invalid RFORMAT command:------	The format descriptor line you entered isn't acceptable. Check the spelling and make sure that a tab delimits each field.
Not a valid command line	The Report Writer doesn't recognize a format descriptor line. Check the syntax. All commands must be delimited by a tab.

Error Message	Cause
There MUST be a detail, total, or break line somewhere.	You tried to define an rformat without at least one DETAIL line, one BREAK line, or one TOTAL line. The Report Writer won't accept it.
Too many arguments on one report command line	You used more arguments than allowed. Check the syntax.
Field starting location is before beginning of line -----	You'll get this error message or one of the two below it if the item(s) on the named line do not fit properly. Check the syntax. For instance, remember that an unpictured number requires a 14 character field.
A field spreads across the right end of -----	
Overlapping items on -----	
Total Errors:	You've got the displayed number of errors
Wrong number of arguments for this command: -----	Check what you typed. You may have used blanks instead of tabs or the wrong number of arguments in your rformat. Fix them.

BREAK Statement

Error Message	Cause
Item does not exist:-----	The operand you specified is not a data item, a defined item, or a literal. These are your only choices.
Invalid BREAK or TOTAL expression: -----	The Report Writer doesn't understand what BREAK or TOTAL actions you want performed. Check the syntax and/or for typos.
Invalid PRINT_OP expression:-----	The Report Writer can't understand the expression. If you want to underline, parentheses must surround the named item. Note, also, that in a PRINT_OP expression, a blank separates the keyword from the argument.
Line location is invalid: -----	You specified an invalid location for the BREAK information. You can use either a positive or negative number, or C, for centered.
Not a Data_item: -----	The Report Writer did not recognize the specified item. Check for typos, missing or misplaced tabs, and make sure that your qformat command worked. If you intended the item to be a literal, did you put quotation marks around it?
This field not yet used in a DETAIL line: -----	You wanted an item underlined, but you haven't named that item on a DETAIL line yet. Be sure the DETAIL statement for that item precedes the BREAK statement for it.
Too many unique BREAK items requested	You tried to define too many BREAK actions. You can define BREAK actions for up to 5 data items in the rformat.

COLUMNS PER LINE Statement

Error Message	Cause
Invalid number: ----	You've used something other than a number as an argument; the COL/LIN statement requires a number as an argument.
Multiple CPL (characters per line) definitions	You tried to use more than one COL/LIN statement in the rformat. Only one is allowed.
Number too large: -----	You used a number as an argument, but it was too large. You can indicate up to 132 characters to a line in the COL/LIN statement.

DEFINE Statement

Error Message	Cause
Item already exists:-----	You tried to redefine an item that has already been defined. Use a different variable name.
Item does not exist:-----	The operand you specified is not a data item, a defined item, or a literal. These are your only choices.
The expression is missing an operator	The Report Writer is looking for an operator separated from the two operands by spaces. Tabs will not work here.
Too many DEFINE items specified	You tried to DEFINE too many items. You're limited to 10 DEFINE variables per rformat.

DETAIL Statement

Error Message	Cause
Invalid number: -----	You'll get this message or the one below it if you specify an invalid DETAIL line number. Try an integer between 1 and 5.
Line # is too great: -----	
Item does not exist: -----	The Report Writer doesn't recognize the named item. It is not a known item or a literal.
Line location is invalid:-----	You specified an invalid location for the starting position of some DETAIL information. You can use a positive or negative number, or C, for centered.

HEADER Statement

Error Message	Cause
Invalid number: ----- Line # is too great: -----	You'll get this error message or the one below it if you specify an invalid HEADER line number. Try an integer between 1 and 10.
Line location is invalid	You specified an invalid location for the starting position of some HEADER information. You can use a positive or negative number, or C, for centered.
Not a Data_item: -----	You specified a data item that has not yet been defined in a qformat. Try a different data item, or define this one in a qformat.

LINES PER PAGE Statement

Error Message	Cause
Invalid number: ----- Too few lines per page specified.	Either of these messages means you specified an invalid number of lines. The minimum allowed is 30.
Multiple LPP (lines per page) definitions	You tried to use more than one LIN/PG statement in the rformat. Only one is allowed.

PICTURE Statement

Error Message	Cause
Invalid characters in expression	You used nonpicture characters in your Picture clause.
Invalid PICTURE format:-----	You used invalid syntax in your Picture clause. Check the syntax.
Item does not exist: -----	You specified a picture clause for an item that doesn't exist. The named item must be a data or defined item. Try again.
Multiple PICTURE definitions for one item: -----	You can specify only one PICTURE per item. Choose the PICTURE which best fits. If necessary, DEFINE another item in terms of the first and then specify the PICTURE it needs.
PICTURE definition is too long	You tried to use more than 30 characters.
Too many digits to the left of the decimal Too many digits to the right of the decimal	You'll get this error message or the one below it if you use too large a Picture clause. You are allowed 12 characters to the left of the decimal point and 4 to the right.

QFORMAT Statement

Error Message	Cause
Multiple qformat definitions	You specified more than one qformat in the rformat. Only one is allowed.
---x--- is not a defined format	You've specified a qformat that is not in the .QFORMS file, has syntax errors, or you've misspelled the name.

SORT Statement

Error Message	Cause
You may not have SORT lines in the rformat	Sort/Merge's Report Writer doesn't accept INFOS II QUERY with Report Writer SORT lines. Sort your data by using Sort/Merge command file statements.

TOTAL Statement

Either of these messages means you specified an invalid TOTAL line number. Try an integer between 1 and 10.

Error Message	Cause
Invalid number: ----- Line # is too great: -----	You'll get this error message or the one below it if you specify an invalid TOTAL line number. Try an integer between 1 and 10.
Invalid TOTAL expression:-----	The Report Writer assumes that you want total information displayed from the first three fields of the TOTAL statement, but can't interpret the fourth field. Check it out and try again.
Item does not exist: -----	The Report Writer doesn't recognize the named item. Try either a data or defined item.
Line location is invalid:-----	You specified an invalid location for the TOTAL information. You can use either a positive or negative number, or C, for centered.

End of Appendix

Appendix D

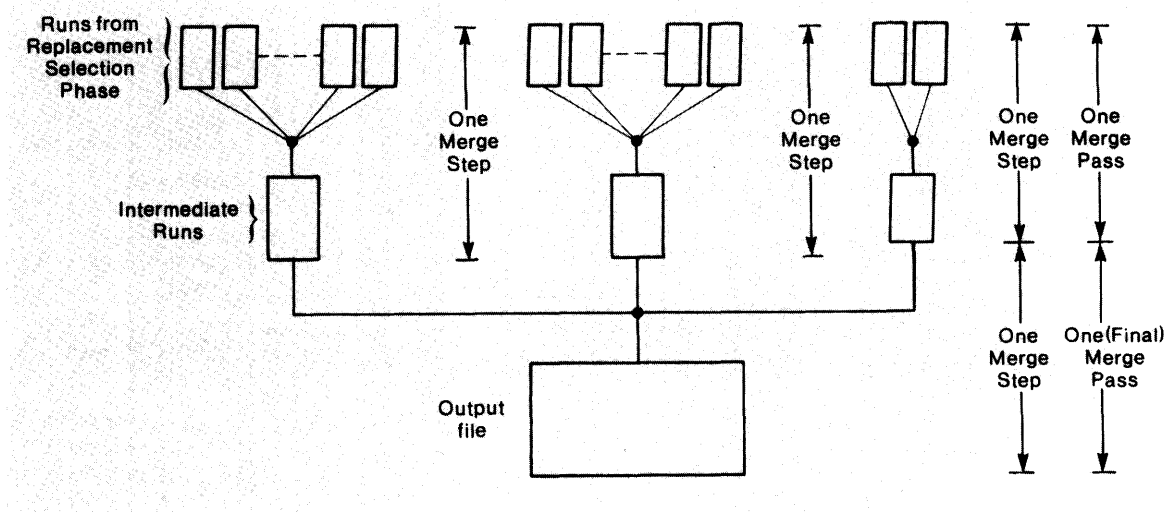
Statistical Information the Utility Returns

By default, the utility displays statistics for each command file execution. If you don't want to see the statistics, include the /S switch in the command line (see Chapter 7).

In the statistics, you'll find terms which we briefly define here.

Term	Definition
Replacement selection	Algorithm which sorts
Run	Sequence of records in sorted order
Merge Step	The combining of many runs into one run
Merge pass	A sequence of one or more merge steps which results in one or more runs

Figure D-1 shows an example of conceptually how the utility merges runs (in the process of sorting). In the first merge pass, the utility combines runs from its replacement selection phase. Here, the utility combines the runs in three merge steps. These three steps leave three intermediate runs. In the second (and final) merge pass, the utility combines the intermediate runs in one merge step. This merge step produces the final output file.



SD-02367

Figure D-1. Runs, Steps, and Merge Passes

Table D-1. Statistics Produced for Each Operation Phase

During this Phase of the Utility's Operation	The Utility Performs these Operations	And Generates these Statistics
Setup and Validation	Verifies that there are no syntactical errors in the command line and/or command file. Sets up the modules it needs to execute the requested actions.	SETUP AND VALIDATION PHASE TIME OUTPUT FILE RECORD FORMAT
Input	Reads the input records and massages them as specified. The utility may skip certain input records, depending on any input IF statements.	TOTAL NUMBER OF INPUT RECORDS MINIMUM INPUT RECORD LENGTH MAXIMUM INPUT RECORD LENGTH TOTAL NUMBER OF SKIPPED RECORDS
Replacement Selection	Sorts the input records passed to this phase, which produces runs. The utility passes runs to the merge phase as input.	REPLACEMENT SELECTION PHASE TIME TOTAL NUMBER OF OUTPUT RUNS RECORDS PASSED TO THE MERGE SIZE OF SELECTION TREE BIAS FACTOR

(continues)

Table D-1. Statistics Produced for Each Operation Phase

During this Phase of the Utility's Operation	The Utility Performs these Operations	And Generates these Statistics
Merge	Merges the contents of the runs produced in the replacement selection phase. Massages records as specified. The utility may skip records depending on the presence of any output IF massage statements or a DELETING DUPLICATES imperative.	MAXIMUM INTERMEDIATE ORDER OF MERGE
		MERGE WILL REQUIRE MULTIPLE PASSES
		NUMBER OF STEPS IN FIRST PASS
		PER PASS MERGE TIME
		RECORDS WITH DUPLICATE KEYS
		TOTAL SKIPPED RECORDS (MERGE PHASE)
		USER (FINAL) MERGE ORDER
Output	Writes the records to the output file	(FINAL) NUMBER OF RECORDS OUTPUT
		MIMINUM OUTPUT RECORD LENGTH
		MAXIMUM OUTPUT RECORD LENGTH
		PAGES WRITTEN TO REPORT FILE
		TOTAL ELAPSED TIME

(concluded)

The statistical output is in decimal integers. The utility rounds fractions to the next higher integer. The statistics and their meanings are

SETUP AND VALIDATION PHASE TIME

The number of seconds it takes the utility to

- verify that there are no syntactical errors in the command line and/or command file
- set up its modules to execute the functions you've requested

OUTPUT FILE RECORD FORMAT

The record type of the output file. If the output file does not exist before the utility executes the command file, the utility creates the output file.

(TOTAL) NUMBER OF INPUT RECORDS

The sum of all records that the utility encounters in all input files.

MINIMUM INPUT RECORD LENGTH

The length in characters of the shortest input record the utility encounters.

MAXIMUM INPUT RECORD LENGTH

The length in characters of the longest input record the utility encounters.

(TOTAL) NUMBER OF SKIPPED RECORDS

The number of records the utility does not write to the output file, due to IF statements in which you specify the SKIP option.

REPLACEMENT SELECTION PHASE TIME

The number of seconds it takes the utility to make the first sorting pass over the input records.

TOTAL NUMBER OF OUTPUT RUNS

The total number of runs that the replacement selection phase produces.

RECORDS PASSED TO THE MERGE

The total number of records in all runs that the replacement selection phase produces.

SIZE OF SELECTION TREE

The number of records held in memory at one time during the initial sorting pass.

BIAS FACTOR

A measure of how well input file records are sorted before you use the utility. It takes a large number of records to produce this statistic.

A bias factor of	**means input file records are**
1	in reverse order from the order that you want
approximately 2	randomly ordered
greater than 2	nearly sorted

MAXIMUM INTERMEDIATE ORDER OF MERGE

The largest number of runs that the utility merges to form a single run.

MERGE WILL REQUIRE MULTIPLE PASSES

You declared so many input files for merging that the utility requires multiple merge passes.

NUMBER OF STEPS IN FIRST PASS

Unless the intermediate merge is for a stable sort, the utility attempts to economize the work it does on the first pass. If the number of steps is small, this statistic reflects the success of the utility's attempt.

PER PASS MERGE TIME

The time the utility takes to complete one merge pass over the runs in the work file.

RECORDS WITH DUPLICATE KEYS

The number of records discarded from output because they have duplicate keys. You receive this statistic if you use the DELETING DUPLICATES imperative.

TOTAL SKIPPED RECORDS (MERGE PHASE)

The number of records the utility does not place into an output run due to IF massage statements in which you included the SKIP option for output file massaging.

USER (FINAL) MERGE ORDER

This statistic is either

- the number of runs that the utility combines in the final merging pass of a sort process, or
- the number of files which the utility combines in one merge step of a merge process

(FINAL) NUMBER OF RECORDS OUTPUT

The actual number of records the utility writes to the output file.

MINIMUM OUPUT RECORD LENGTH

The length in bytes of the shortest record the utility writes to the output file.

MAXIMUM OUTPUT RECORD LENGTH

The length in bytes of the longest record the utility writes to the output file.

PAGES WRITTEN TO REPORT FILE

The number of line-printer pages created by printing a report file.

TOTAL ELAPSED TIME

The actual amount of time (real time) that elapses from when you invoke the utility to when the utility writes the last output record.

THERE WERE n RUNTIME ERRORS DURING REPORT GENERATION

The number (n) of runtime errors encountered during report generation. This statistic won't appear if n=0.

End of Appendix

Appendix E

Faster Sorts and Merges: Fine Tuning the Utility

There are a few procedures that you can follow to improve the utility's performance. These procedures range in difficulty from those that are easy, to those that only a very sophisticated AOS user would perform. We'll discuss the procedures in this ascending order of difficulty.

Optimal Record Length Estimate

You can use the following procedure when your input files have variable-length or data-sensitve records. If you know the length of the longest record, specify that as integer in the

- VARIABLE UPTO phrase of the AOS INPUT FILE, and/or INPUT INFOS declarations, or
- DATA SENSITIVE phrase of the AOS INPUT FILE

You may not know the exact length of the longest record. In that case, determine a length which you think no record will exceed and use that length as integer. The more closely you estimate the length of the longest record, the more you'll improve the utility's performance. For example, if you think that the longest data-sensitive record in a file is about 190 characters, you could let integer equal 200.

Increase Element Size

You'll probably improve the utility's performance if you create a work file with a large element size. The larger the element size, the fewer the elements AOS needs for a given file. Fewer elements is better because the more elements a file has, the more index blocks AOS needs to keep track of and access the elements in the file. This of course slows down AOS, which in turn slows down Sort/Merge.

The best work file is a contiguous file. If you create the work file's element size large enough so that the file requires only one element, then you guarantee that the file will be contiguous. Note, however, that the largest contiguous file AOS can create is 65,534 blocks. A file of this size can serve as a work file for a sort of no more than 25 million bytes of data. Also, due to disk fragmentation, it's almost impossible to find very large elements unless the disk is freshly formatted or contains very few files.

You must create the work file in order to select its element size. You create a work file with this CLI command:

```
CREATE/ELEMENTSIZE=integer filename
```

Argument integer is the decimal number of disk blocks (512 bytes each) per element.

If you cannot sort with contiguous work files, a file structure with a single index level is almost as efficient. As a rule of thumb, determine the element size necessary for a single-level index work file by dividing the number of bytes in the input file by 50,000. We restate this rule of thumb as a formula:

Element Size = (Number of Bytes in Input File)/50,000

If you create your own work file, you'll specify its name in a WORK FILE declaration. When you do so, the utility uses that work file instead of creating its own. Also, the utility will not delete the work file that you create when execution finishes.

You can also increase the element size (from the default value of 1) of a large AOS output file to improve the utility's performance. The AOS OUTPUT FILE declaration's ELEMENTS ARE clause lets you select the element size. Even a modest increase in element size (4, 8, or 16) will improve performance. For best results, apply the rule of thumb formula and write to a single-level index output file.

Optimal File Placement

This section explains how to optimize the utility's performance based on your system's configuration of peripherals. Before you continue reading this section, you should know that

- we assume you know about AOS disk structures
- we use some terms defined in the introduction to Appendix D

All the procedures point toward one goal. You want to reserve each device only for reading or only for writing operations during a pass of the utility. This allows I/O to proceed more quickly because an individual disk will not have to do time-consuming seek operations to alternately access input and output files. (You lose this advantage to some extent if another process is accessing another file on the same device.) To reserve the devices, you must have access to directories on more than one Logical Disk Unit (LDU). In general, you want to place different files on separate LDUs.

In the discussion below, we consider three basic system configurations. The first system has three disk drives, the second has two disk drives, and the third has one. We also consider the effects of adding a tape drive to the two and one disk configurations. All cases assume that that you have created the work files with an appropriately large element size (discussed in the previous section). The discussions about the different configurations sometimes refer to Figure E-1.

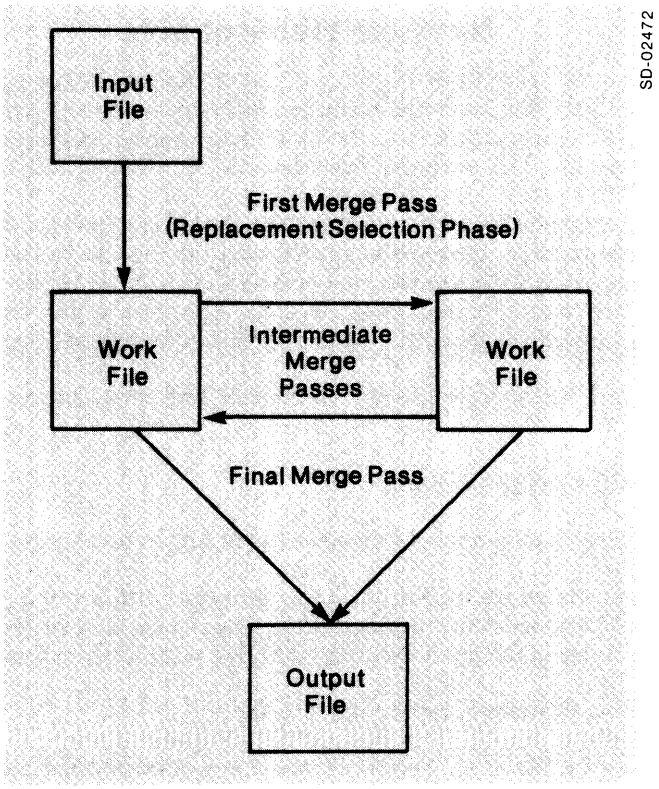


Figure E-1. Flow of Data

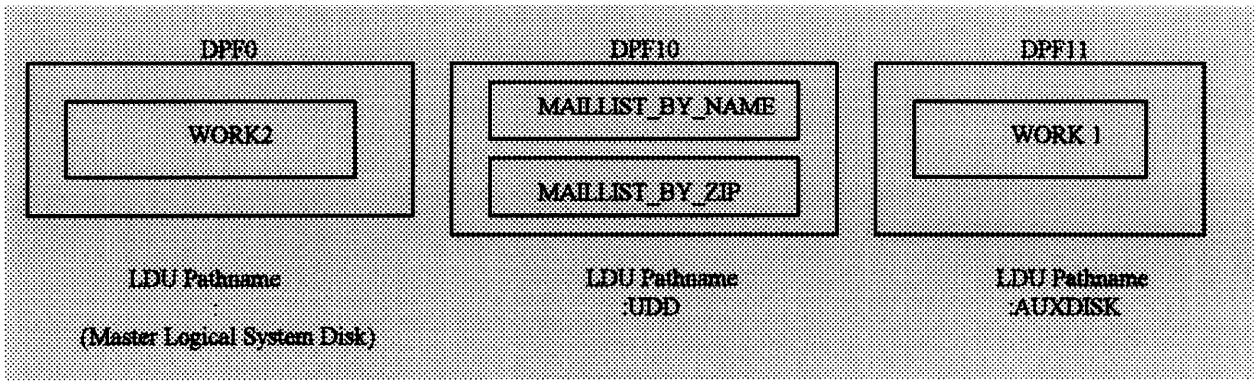


Figure E-2. File Placement on Three Disk Drives

Let's follow the flow of data during the execution of this sort. By doing this, you'll see that each device is reserved only for reading or only for writing operations during one pass of the utility. On the first pass (replacement selection), Sort/Merge reads input file records on :UDD and writes those records to the work file on :AUXDISK. During the intermediate merge passes, Sort/Merge reads records from the :AUXDISK work file and writes records to the : (master logical system disk) work file, or vice versa. On the final pass, Sort/Merge reads records from both the : and :AUXDISK work files. Then it writes those records to the output file on :UDD.

Two Disk Drives

Figure E-3 shows file placement on two disk drives. Notice that we placed the work files on different disks. Also note that work file WORK1 is not on the same disk as the input file.

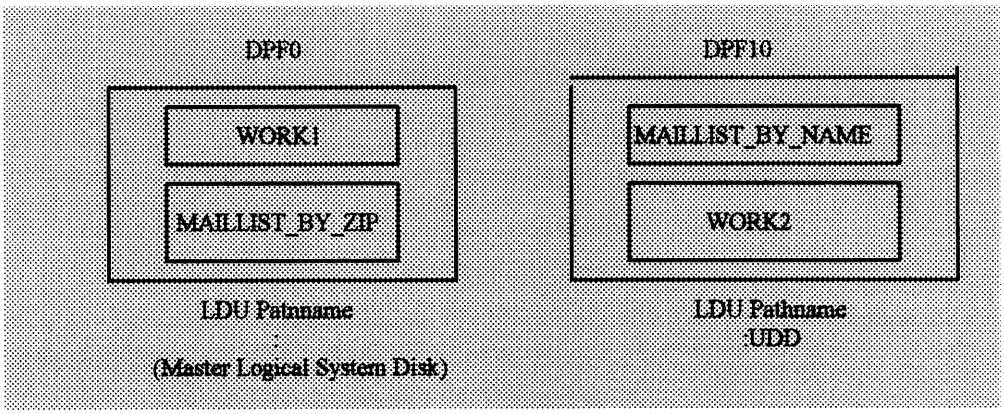


Figure E-3. File Placement on Two Disk Drives

The optimal command file which uses this arrangement looks like this:

```
INPUT FILE IS ':UDD:USER:MAILLIST_BY_NAME".
OUTPUT FILE IS  ":USER:MAILLIST_BY_ZIP",
          ELEMENTS ARE 32 BLOCKS.
KEY 102/106.
WORK FILE IS   ":USER:WORK1".
WORK FILE IS ":UDD:USER:WORK2".
SORT.
END.
```

Figure E-1 shows the flow of data in the sorting process. The first merge pass (during the replacement selection phase) moves data from the input file(s) to a work file. Intermediate merge passes move data from from one work file to another. The final merge pass moves data from the work files to the output file.

We use one example for the three configurations. In this example, you want to sort a mailing list by zip code. The input file is MAILLIST_BY_NAME and the output file is MAILLIST_BY_ZIP. The two work files are WORK1 AND WORK2.

Three Disk Drives

Figure E-2 shows file placement on three disks drives. Notice that we placed the two work files on different disks. The optimal command file which uses this arrangement looks like this:

```
INPUT FILE IS ":UDD:USER:MAILLIST_BY_NAME".
OUTPUT FILE IS ":UDD:USER:MAILLIST_BY_ZIP",
    ELEMENTS ARE 32 BLOCKS.
KEY 102/106.
WORK FILE IS ":AUXDISK:USER:WORK1".
WORK FILE IS ":USER:WORK2".
SORT.
END.
```

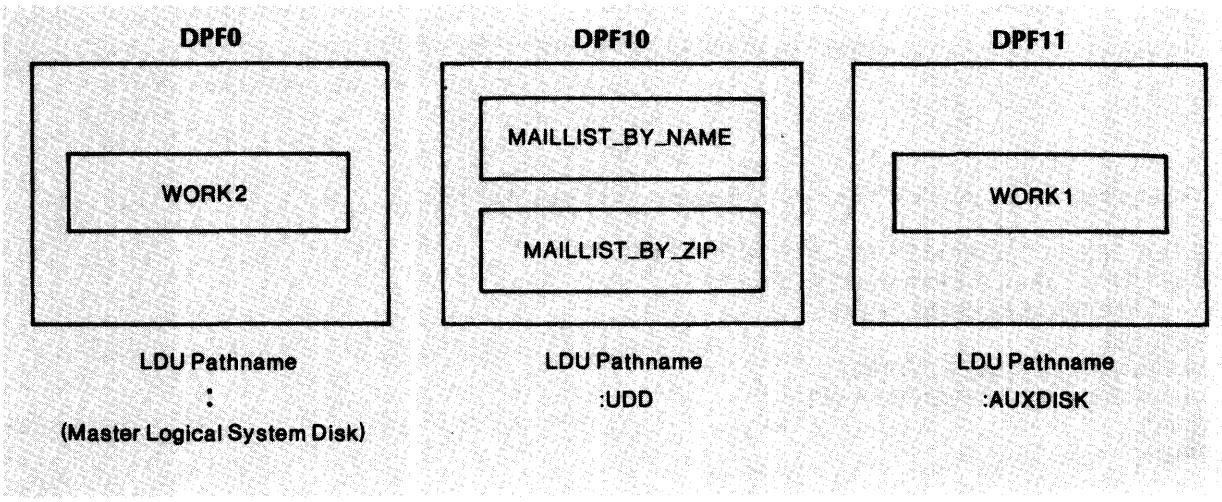


Figure E-2. File Placement on Three Disk Drives

Let's follow the flow of data during the execution of this sort. By doing this, you'll see that each device is reserved only for reading or only for writing operations during one pass of the utility. On the first pass (replacement selection), Sort/Merge reads input file records on :UDD and writes those records to the work file on :AUXDISK. During the intermediate merge passes, Sort/Merge reads records from the :AUXDISK work file and writes records to the : (master logical system disk) work file, or vice versa. On the final pass, Sort/Merge reads records from both the : and :AUXDISK work files. Then it writes those records to the output file on :UDD.

Two Disk Drives

Figure E-3 shows file placement on two disk drives. Notice that we placed the work files on different disks. Also note that work file WORK1 is not on the same disk as the input file.

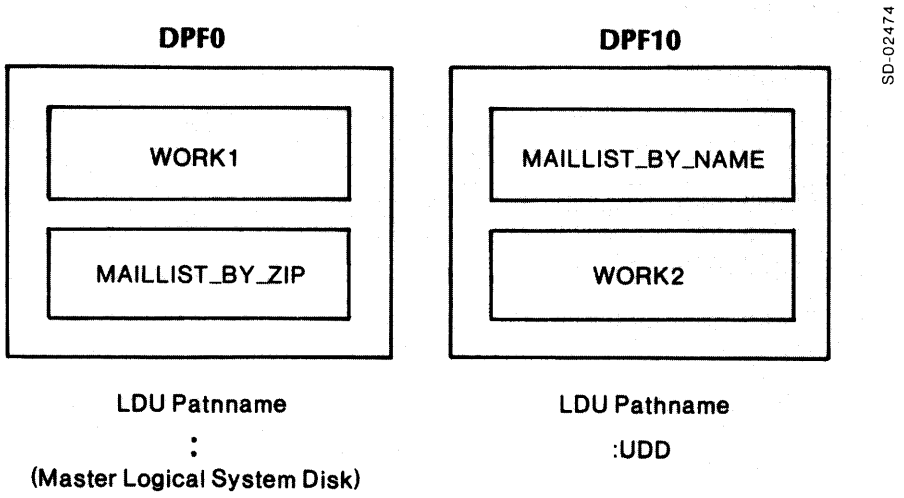


Figure E-3. File Placement on Two Disk Drives

The optimal command file which uses this arrangement looks like this:

```
INPUT FILE IS ":UDD:USER:MAILLIST_BY_NAME".
OUPUT FILE IS ":USER:MAILLIST_BY_ZIP",
   ELEMENTS ARE 32 BLOCKS.
KEY 102 / 106.
WORK FILE IS ":USER:WORK1".
WORK FILE IS ":UDD:USER:WORK2".
SORT.
END.
```

Let's follow the flow of data during the execution of this sort. On the first pass, data moves from :UDD to : (master logical system disk). During the intermediate merges, data moves back and forth between the two disks. On the final pass, the optimal situation no longer holds; reading and writing occur on the same disk, :UDD, during the same pass.

You can make a sort which uses two disks more efficient. In most cases, you can write the final output to a tape file. Figure E-4 shows this configuration. This usually has two advantages:

1. on the final pass, Sort/Merge only reads from the work file disks and only writes to the tape file

2. you avoid the operating system overhead in allocating disk elements

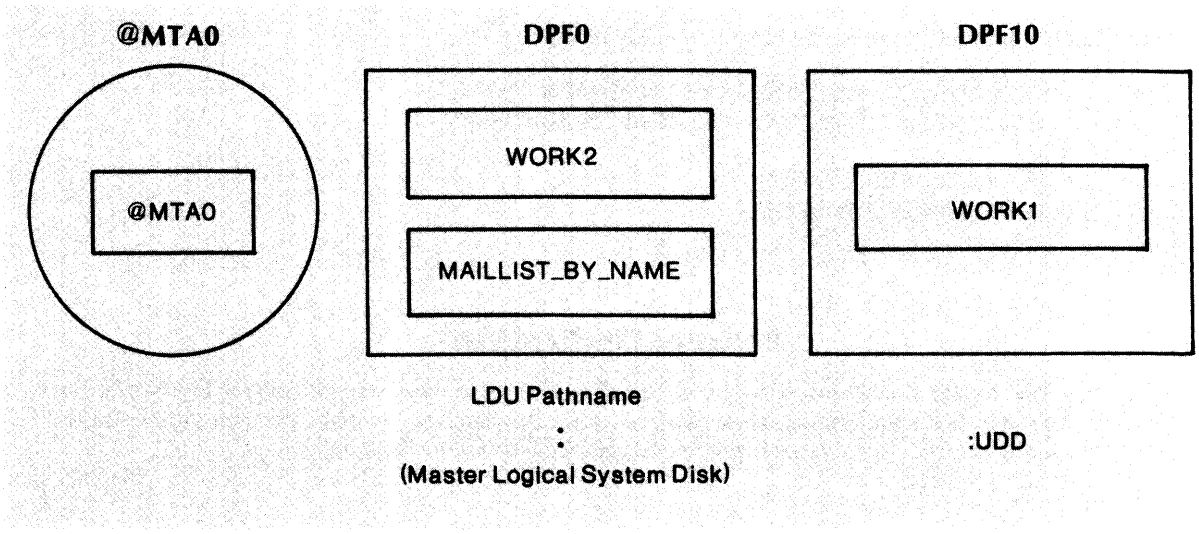


Figure E-4. File Placement on Two Disk Drives and One Tape Drive

One Disk Drive

You obviously must place all files on one disk if you have only one disk. Thus, there's no point in discussing optimal file placement on one disk, unless you include a tape drive with the disk. Figure E-5 shows file placement on one disk and a tape drive. The figure shows both the input and output files on the tape drive. You could put just the input or just the output file on the tape drive.

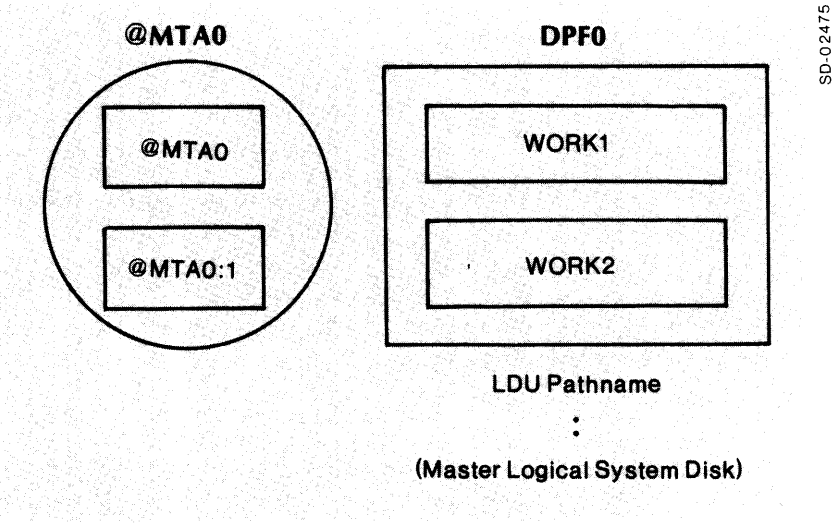


Figure E-5. File Placement on One Disk and One Tape Drive

The optimal command file using this arangement looks like this:

```
INPUT FILE IS "@MTB0:0", RECORDS ARE 200 CHARACTERS.
OUTPUT FILE IS "@MTB0:1, BLOCKS ARE 4096 CHARACTERS.
KEY 102/106.
WORK FILE IS ":UDD:USER:WORK1".
WORK FILE IS ":UDD:USER:WORK2".
SORT.
END.
```

Process Dedication

Competing disk traffic slows Sort/Merge considerably. This is because the processing of several files scattered around a disk pack also causes seeking. If you can, dedicate the disks the utility needs to the utility. Try not to let other jobs make frequent competing accesses to these disks.

End of Appendix

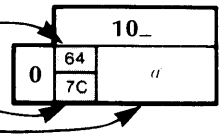
Appendix F

ASCII Character Set

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character



OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS (BACK- SPACE)	16 10	DLE ↑P	24 18	CAN ↑X	32 40	SPACE	40 4D	(	48 F0	0	56 F8	8
1	1 01	SOH ↑A	9 05	HT (TAB)	17 11	DC1 ↑Q	25 19	EM ↑Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	STX ↑B	10 15	NL (NEW LINE)	18 12	DC2 ↑R	26 3F	SUB ↑Z	34 7F	" (QUOTE)	42 5C	*	50 F2	2	58 7A	:
3	3 03	ETX ↑C	11 0B	VT (VERT. TAB)	19 13	DC3 ↑S	27 27	ESC (ESCAPE)	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	EOT ↑D	12 0C	FF (FORM FEED)	20 3C	DC4 ↑T	28 1C	FS ↑\	36 5B	$	44 6B	, (COMMA)	52 F4	4	60 4C	<
5	5 2D	ENQ ↑E	13 0D	RT (RETURN)	21 3D	NAK ↑U	29 1D	GS ↑]	37 6C	%	45 60	-	53 F5	5	61 7E	=
6	6 2E	ACK ↑F	14 0E	SO ↑N	22 32	SYN ↑V	30 1E	RS ↑↑	38 50	&	46 4B	. (PERIOD)	54 F6	6	62 6E	>
7	7 2F	BEL ↑G	15 0F	SI ↑O	23 26	ETB ↑W	31 1F	US ↑←	39 7D	' (APOS)	47 61	/	55 F7	7	63 6F	?

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	` (GRAVE)	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91 8D	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 C5	E	77 D4	M	85 E4	U	93 9D	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ (TILDE)
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or _	103 87	g	111 96	o	119 A6	w	127 07	DEL (RUBOUT)

SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Data General

TP_____

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
- ☐ U.P.S. Blue Label
- ☐ Air Freight
- ☐ Other _____

_____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order _____ Phone _____ Extension _____

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

_____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

_____
Title

_____
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

DataGeneral

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!