Chapter 6
DATA DIVISION

The DATA DIVISION describes the data that the object program uses, creates, and produces as output. This chapter defines the physical limitations that apply to the DATA DIVISION, the types of data-items that may be specified in a Microsoft COBOL program with the USAGE clause, and the section headers and clauses that provide the structure for your data.

The last two parts of the chapter present the individual sections and clauses that are used in the DATA DIVISION. Section 6.4, "Sections," presents the sections, in the order in which they appear in a source program, and Section 6.5, "Clauses," presents the clauses that make up these sections, in alphabetical order.

6.1 DATA DIVISION Header and General Format

Purpose

Describes the data that will be used, created, or produced by the object program.

Format

The DATA DIVISION contains four sections:

FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION
SCREEN SECTION

These sections are discussed individually in Section 6.4, "Sections," of this chapter. The SCREEN SECTION is a Microsoft extension for interactive screen-handling.

The general format for the DATA DIVISION is:

```
DATA DIVISION.

[ FILE SECTION.

    [ file-description-entry { record-description-entry } ...
    | sort-merge-file-description-entry { record-description-entry } ... ] ...

[ WORKING-STORAGE SECTION.

    [ 77-level-description-entry
    | record-description-entry ] ... ]

[ LINKAGE SECTION.

    [ 77-level-description-entry
    | record-description-entry ] ... ]
```

```
[SCREEN SECTION.
{level-number [ screen name ]
        [ BLANK SCREEN ]
        [ LINE NUMBER IS [ PLUS ] integer-1 ]
        [ COLUMN NUMBER IS [ PLUS ] integer-2 ]
        [ FOREGROUND-COLOR integer-3 ]
        [ BACKGROUND-COLOR integer-4 ]
        [ BLANK LINE ]
        [ BELL ]
        [ UNDERLINE ]
        [ REVERSE-VIDEO ]
        [ HIGHLIGHT ]
        [ BLINK ]
        { [ [ VALUE ] IS literal-1 ]
        [ {PICTURE} IS character-string { [ FROM {literal-2   } ] [ TO identifier-2 ] } ] }
        [ {PIC    }                     { [ USING identifier-3 ]                      } ]
        [ BLANK WHEN ZERO ]
        [ {JUSTIFIED} RIGHT ]
        [ {JUST     }       ]
        [ AUTO ]
        [ SECURE ]
        [ REQUIRED ]
        [ FULL ]. } ...]
```

Remarks

The DATA DIVISION is a required part of an MS-COBOL program. It describes the data that were listed in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. It also describes data used in the program that are not part of the input-output sections of the program (i.e., that are in the WORKING-STORAGE, LINKAGE, and SCREEN SECTIONS).

These data are arranged in logical records. A logical record can be further divided into fields, or data-items. For example, an "Inventory-Master-File" declared in a FILE-CONTROL paragraph could contain one record for each piece of equipment inventoried. Each record could be further divided into data-items such as PART-NUMBER and DATE-ACQUIRED.

Example

```
DATA DIVISION.

FILE SECTION.

FD  INVENTORY-MASTER-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "MASTER.DAT".

01  MASTER-RECORD.
    05  MSTR-KEY              PIC X(10).
    05  MSTR-DESCRIPTION      PIC X(25).
    05  MSTR-AMT-ON-HAND      PIC S9(5).
    05  MSTR-WARNING-LEVEL    PIC S9(5).

FD  INVENTORY-WARNING-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "WARNING.DAT".

01  WARNING-RECORD            PIC X(45).

FD  INVENTORY-REPORT-FILE
    LABEL RECORDS ARE OMITTED
    LINAGE IS 56 LINES.

01  REPORT-RECORD             PIC X(80).

WORKING-STORAGE SECTION.

01  WORK-FIELDS.
    05  MASTER-STATUS         PIC XX
        VALUE SPACES.
    05  WARNING-STATUS        PIC XX
        VALUE SPACES.
    05  REC-COUNT             PIC S9(5)
        VALUE ZERO.
    05  WARNING-COUNT         PIC S9(5)
        VALUE ZERO.
    05  END-OF-FILE-SW        PIC X
        VALUE "N".
        88  END-OF-FILE
            VALUE "Y".
```

6.2 Record Description Entry

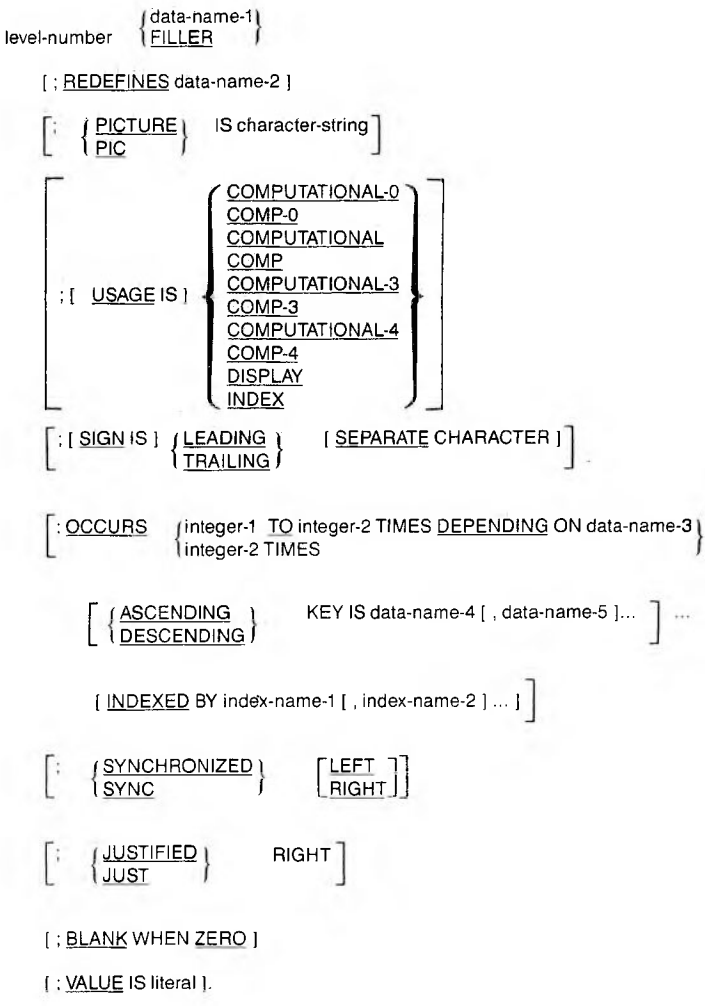
A record description entry includes all the data description entries for that record (see the following pages for the complete syntax).

Within the DATA DIVISION, record-items and data-items are given level numbers. Record-items are given level-numbers of 01. Data-items (fields within records and other data-items needing storage by the object program) are given level-numbers 02 through 49 and are declared in "data description entries."

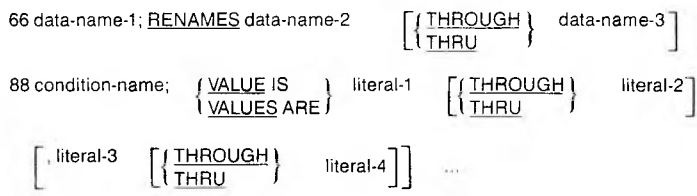
For naming purposes, records are considered as data-items and follow the rules given for data-names in Chapter 2, "Language Elements."

Level 66, 77, and 88 entries may also be used in the DATA DIVISION. Level 66 entries support the RENAMES clause which regroups data-items. Level 77 entries describe elementary data-items that are not subordinate to any other data-item. Level 88 entries describe conditional variables.

The general format for a record or data description entry (including level 77 entries) is:



or, in the case of level 66 and 88 entries, respectively:



The clauses used in a record description entry may appear in any order, except that the data-name must immediately follow the level-number, and if the REDEFINES clause is used, it must immediately follow the data-name.

The PICTURE clause is required in every elementary data-item description except an index-data-item.

The clauses PICTURE, JUSTIFIED, SYNCHRONIZED, and BLANK WHEN ZERO can only be specified for elementary data-items.

For a discussion of the restrictions on the use of level 66, level 77, and level 88 data-items, see Sections 6.2.7, 6.2.8, and 6.2.9, respectively.

6.2.1 Data Description Entries and Data-Items

Each data-item in the object program (for example, fields within records and other data-items needing storage by the object program) must be described in a separate record description or data description entry. These descriptions are entered in the DATA DIVISION and are given level-numbers 02 through 49. For convenience, we will generally use the term "data description entry" to mean both record description entry and data description entry.

Note, however, that a record description entry is a specific type of data description entry, and always refers to data that begins with level-number 01. We will use the term "record description entry" when the data to be described must begin with level-number 01.

Data-items must be entered in the order in which the items appear in the record, and can be either group items having subordinate data elements, or elementary items, which do not have subordinates. Elementary items can be further classified and defined by their content (see Section 6.2.3, "Elementary Items").

The following discussion defines group and elementary data-items and the elementary data-item types supported by Microsoft COBOL.

The general format for a data description entry is given in Section 6.2 of this chapter. Every entry must contain a level number, data-name or the word "FILLER", and a series of clauses, followed by a period (.). Specific types of data-items, however, require certain clauses and cannot contain others. These requirements and restrictions are discussed in Sections 6.2.4 through 6.2.9 of this chapter.

6.2.2 Group Items

A group item is any item that is further subdivided into elementary items or subordinate group items.

Example:

```
01  GROUP-NAME.
     02  FIELD-A PICTURE X.
     02  FIELD-B.
         03  FIELD-C PICTURE X.
         03  FIELD-D PICTURE X.
```

In this example, the level 02 items are subordinate to the level 01 group item. The level 02 group item, FIELD-B, contains two level 03 subordinate elementary items, FIELD-C and FIELD-D. The 01 level number also indicates the beginning of a new record; all items will be part of that record until another 01 level number is encountered. The maximum size of a group item, including its subordinate items, is described in Section 6.3, "DATA DIVISION Limitations."

The following clauses may be used to modify the data description entry:

OCCURS
REDEFINES
SIGN
USAGE

6.2.3 Elementary Items

An elementary level data-item is one that contains no subordinate items, and may be of these data-item types:

1. alphabetic
2. alphanumeric
3. alphanumeric-edited
4. numeric
5. numeric-edited
6. level 66 (RENAMES)
7. level 77 (noncontiguous)
8. level 88 (condition-names)

The default internal storage format for elementary data-items is DISPLAY where the data are stored in standard ASCII format, except for index-data-items and index-names which are stored in COMP-0 (binary) format.

6.2.4 Alphanumeric and Alphanumeric-Edited Items

An alphanumeric item consists of any combination of characters making a "character string" data field.

If the associated PICTURE clause contains any editing characters, the item is an alphanumeric-edited item. This form of a data-item is permissible as a receiving field for numeric data but cannot be used as an arithmetic operand.

The PICTURE clause is required.

The following clauses may be used to modify the data description entry:

JUSTIFIED	SYNCHRONIZED
OCCURS	USAGE
REDEFINES	VALUE

Example:

```
02 MISC-1 PICTURE X(53).
02 MISC-2 PICTURE BXXXBXXB.
```

In this example, MISC-1 may contain any combination of characters, with a maximum of 53 characters. The "B" in MISC-2 is an edit character representing a space.

6.2.5 Numeric Items

A numeric item is an elementary item that contains numeric data only. There are four kinds of numeric items:

1. external decimal items
2. internal decimal items
3. binary items
4. index-data-items and index-names

These classifications are based on how the items are stored in memory.

1. External decimal item

   An external decimal item is one in which each character represents one decimal digit in standard ASCII format. The maximum number of digits that can be represented is 18.

   The exact number of digits in an item is defined by the PICTURE clause in the item. For example, PICTURE 999 defines a three-digit item whose maximum value is nine hundred ninety-nine.

   If the value of PICTURE begins with the letter "S", the item will also contain an algebraic operational sign.

This means that any data stored in the field as the result of a MOVE statement or an arithmetic statement will contain the algebraic sign of the result.

The sign does not occupy a separate byte unless the SEPARATE form of the SIGN clause is used in the general format.

The USAGE of an external decimal item is implicitly DISPLAY.

The PICTURE clause is required with external decimal items.

The following clauses may be used to modify the data description:

OCCURS	SYNCHRONIZED
REDEFINES	USAGE
SIGN	VALUE

Examples:

```
02  HOURS-WORKED PICTURE 99V9
    USAGE IS DISPLAY.

02  HOURS-SCHED PICTURE S99V9
    SIGN IS SEPARATE TRAILING.
```

The "V" in the PICTURE clause represents an implied decimal point, and "S" represents an operational sign.

2. Internal decimal item

   An internal decimal item is one that is stored in packed binary-coded decimal (BCD) format. The USAGE IS COMPUTATIONAL-3 form of the USAGE clause specifies this format.

   An internal decimal item defined by n − 9's in its PICTURE clause occupies (n + 2)/2 bytes in memory (rounded down). All bytes except the rightmost contain a pair of digits, and each digit is represented by the binary equivalent of a valid digit value from 0 to 9. The item's low order digit and the operational sign are stored in the rightmost byte of a packed item. For this reason, the compiler considers a packed item to have an arithmetic sign, even if the original PICTURE clause lacked an S-character.

The USAGE IS COMPUTATIONAL-3 clause is required for an internal decimal item.

The following clauses may be used to modify the data description:

OCCURS	SYNCHRONIZED
REDEFINES	VALUE
SIGN	

Example:

```
05  TAX-RATE PICTURE S99V999
    VALUE 1.375
    USAGE IS COMPUTATIONAL-3.
```

3. Binary item

A binary item uses the base 2 system to represent an integer. Data-items whose USAGE is COMP-0 occupy two bytes of memory and represent integers in the range −32,768 to 32,767. Data-items whose USAGE is COMP-4 occupy four bytes of memory and represent integers in the range of −2,147,483,648 to 2,147,483,647.

The storage format of binary items may vary among implementations of MS-COBOL.

A binary item is specified by the USAGE IS COMPUTATIONAL-0 or COMPUTATIONAL-4 forms of the USAGE clause.

The PICTURE and USAGE IS COMPUTATIONAL-0 or COMPUTATIONAL-4 clauses are required for binary items.

The following clauses may be used to modify the data description:

OCCURS
REDEFINES
SYNCHRONIZED
VALUE

Warning

We strongly advise against redefining COMP-0 or COMP-4 data-items to refer to parts of such a data-item. The order in which bytes are stored may vary among implementations, or may change in later versions of Microsoft COBOL. Such a practice may limit the portability of your software.

Examples:

```
03  YEAR-TO-DATE PICTURE S9(5)
      USAGE IS COMPUTATIONAL-0.

03  LARGE-VALUE  PICTURE S9(9)
      USAGE IS COMPUTATIONAL-4.
```

4. Index-data-item and index-name

   Index-data-names and index-names are used in table handling. An index-name is defined in the INDEXED BY phrase of the OCCURS clause. It is not declared in a separate WORKING-STORAGE SECTION entry. An index-name is associated with the table whose definition contains the OCCURS clause, and it cannot be used with any other table.

   An index-data-item is defined in a data description entry with the USAGE IS INDEX clause. The PICTURE and VALUE IS clauses are not used in the index-data-item definition. An index-data-item is not associated with a particular table.

   Index-data-items and index-names have an implicit USAGE IS COMPUTATIONAL-0 (binary item) clause.

   See Chapter 9, "Table Handling by the Indexing Method," for more information about using index-data-items and index-names.

The following clauses may be used to modify the data description:

BLANK	SIGN
JUSTIFIED	SYNCHRONIZED
REDEFINES	

Examples:

```
05  TABLE-ENTRY OCCURS 10 TIMES
    PIC 9
    INDEXED BY SUB-VAL.

05  SUB-VAL
    USAGE IS INDEX.
```

In the first example, SUB-VAL is implicitly declared as an index-name associated with TABLE-ENTRY. In the second example, SUB-VAL is declared explicitly, as an index-data-item, but is not associated with a particular table.

6.2.6 Numeric-Edited Items

A numeric-edited data-item is a receiving field for numeric values. It cannot be used as a numeric item itself in numeric calculations. For example, it might be a field named SALES-TOTAL where the calculated figure representing total sales is stored.

A numeric-edited item contains only digits and/or special editing characters such as commas and dollar signs. The maximum number of characters is 30, and the maximum number of digits is 18. The following clauses may be used to modify the data description:

BLANK WHEN ZERO	SYNCHRONIZED
OCCURS	USAGE
PICTURE	VALUES
REDEFINES	

Example:

```
02 SALES-TOTAL PICTURE $$$$,$$9.99-.
```

The minus (−) in the PICTURE clause above represents the location of the operational sign of the calculated result. The dollar sign ($) will "float"; i.e., only one dollar sign will appear in the result, one position to the left of the leftmost non-zero digit in SALES-TOTAL.

6.2.7 Level 66 (RENAMES) Items

Level 66 items provide for multiple names for the same data by renaming individual or adjacent data fields, including group and subordinate items. For example, this code fragment renames a subset of fields in a record. See Section 6.5.24, "RENAMES Clause," for more information.

```
02  STOCK-ON-ACCOUNT.
      03  PERISHABLES        PIC 9(8).
      03  SUNDRIES           PIC 9(8).
      03  DRY-GOODS          PIC 9(8).
      03  APPAREL            PIC 9(8).
      03  ROLLING            PIC 9(8).
      03  SECURITIES         PIC 9(8).
66  RETAIL-STOCK RENAMES STOCK-ON-ACCOUNT THRU
      APPAREL.
```

6.2.8 Level 77 (Noncontiguous) Items

Some data-items and constants may not be part of a hierarchical relationship in the program. These items are not grouped into logical records, and they are not subdivided. Instead, they are given level-number 77 and are classed as "noncontiguous elementary items." They are sometimes called "stand-alone items."

Level 77 entries follow the naming conventions and general format for standard data description entries (see Section 6.2.1, "Data Description Entries and Data-Items").

A PICTURE or USAGE IS INDEX clause is required.

Level 77 entries may be used only in the WORKING-STORAGE and LINKAGE SECTIONS.

6.2.9 Level 88 (Condition-Name) Items

A level 88 condition-name entry specifies a value, list of values, or a range of values that an elementary item may assume. If the specified value matches the value of its associated elementary item, the condition is true; otherwise it is false. For example, the elementary item,

```
02  PAYROLL-PERIOD PICTURE IS 9.
```

may be followed by the level 88 entries

```
88  WEEKLY VALUE IS 1.
88  SEMI-MONTHLY VALUE IS 2.
88  MONTHLY VALUE IS 3.
```

In this case, either of the following conditions may be applied:

```
IF MONTHLY
    PERFORM P100-DO-MONTHLY.

IF PAYROLL-PERIOD = 3
    PERFORM P100-DO-MONTHLY.
```

The elementary item associated with a level 88 entry is called the "conditional variable."

A level 88 entry must be preceded either by another level 88 entry (in the case of several condition-names pertaining to an elementary item) or by an elementary item (which may be FILLER). Index-data-items should not be followed by level 88 items.

The general format for a level 88 entry is:

```
88 condition-name; {VALUE IS   } literal-1 [{THROUGH} literal-2]
                   {VALUES ARE}            [{THRU   }          ]

  [, literal-3 [{THROUGH} literal-4]] ... .
               {THRU   }
```

The VALUE IS or VALUES ARE clause is required.

The following clauses may be used to modify the data description:

THROUGH | THRU

For an edited elementary item, the values in a condition-name entry must be expressed as non-numeric (quoted) literals.

A VALUE clause may contain both a series of literals and a range of literals.

The following rules apply to level 88 condition-names:

1. Every condition-name may be qualified by the name of its associated elementary item and that elementary item's qualifiers.
2. A condition-name may be used in the PROCEDURE DIVISION in place of a simple relational condition.
3. A condition-name may pertain to an elementary item that requires subscripts. In this case, the condition-name, when written in the PROCEDURE DIVISION, must be subscripted according to the same requirements as the associated elementary item.
4. The type of literal in a condition-name entry must be consistent with the data type of its conditional variable.

6.3 DATA DIVISION Limitations

Individual data-items, group or elementary, are limited in size to 60K (61,440) bytes.

As a general rule of thumb, with a small number of files with record sizes under 1K (1024) bytes, and with a small number of LINKAGE SECTION parameters, also under 1024 bytes each, a program can normally have 50 to 60K bytes of working storage. Each new file and linkage section parameter will make at least 1024 bytes of working storage unavailable.

Specifically, the DATA DIVISION in MS-COBOL programs is limited by the equation:

$$w/1024 + \sum_{i=1}^{f} (R_i/1024) + \sum_{i=1}^{l} (D_i/1024) \leq 60$$

where:

w	=	the size of the working storage area, in bytes
f	=	the number of FD and SD entries in the FILE SECTION
R(i)	=	the size of the largest record for each file, in bytes
l	=	the number of 01 or 77 level data-items in the LINKAGE SECTION
D(i)	=	the size of each 01 or 77 level data-item in the LINKAGE SECTION

Note that all the results of divisions are rounded UP to the next highest integer value.

The memory available to the DATA DIVISION is allocated in segments of 1024 bytes. The equation above relates the various allocation restrictions and the maximum number of 1024-byte segments. Note that these segments may overlap when

there are unused portions, and that actual memory used will be less than the number of segments allocated.

Memory assigned to the WORKING-STORAGE area is allocated enough memory segments to contain all data-items. In the equation above, the term

w/1024

reflects the number of segments allocated to the WORKING-STORAGE SECTION.

Memory allocated to file records is similarly allocated enough segments to contain the largest data record described for each file. However, each file is always allocated a new memory segment. Thus, the term

$$\sum_{i=1}^{f} (R_i/1024)$$

reflects the sum of the memory segments required for each individual file record area.

While LINKAGE SECTION items reserve no actual space, they are treated as if memory is actually allocated to them, under the same rules as WORKING-STORAGE. In addition, each level 01 or 77 data-item present is assigned a new 1024-byte segment. Thus, the term

$$\sum_{i=1}^{l} (D_i/1024)$$

represents the sum of the segments allocated to each individual level 01 or 77 LINKAGE SECTION item.

Example:

```
DATA DIVISION.
FILE SECTION.
FD  FILE-1
      .
      .
01  FILE-1-RECORD.
    05  FILE-1-DATA              PIC X(80).
FD  FILE-2
      .
      .
01  FILE-2-RECORD.
    05  FILE-2-DATA              PIC X(2000).

WORKING-STORAGE SECTION.
01  ITEM-1                      PIC X(80).
01  LARGE-ITEM.
    03  LARGE-VALUES OCCURS 100 TIMES.
        05   LARGE-ELEMENT-1        PIC 9(10).
        05   LARGE-ELEMENT-2        PIC 9(10).
        05   LARGE-ELEMENT-3        PIC 9(10).
        05   LARGE-ELEMENT-4        PIC 9(10).

LINKAGE SECTION.
01  PARAMETER-1                 PIC X(10).
01  PARAMETER-2                 PIC X(2000).
77  PARAMETER-3                 PIC 9(5).
```

In this example, we calculate each of the three terms of the DATA DIVISION limitation as follows:

1. The size of WORKING-STORAGE is 80 + (100 * 40), that is, the size of ITEM-1 (80 bytes), plus the size of LARGE-ITEM (100 elements of 40 bytes each). The result, 4080, is divided (rounding up) by 1024, resulting in four 1024-byte segments being allocated.

2. The allocation of file record space is calculated as the size of each file's largest record, divided by 1024, rounded up. For the two files in our example, 80/1024 gives us 1, and 2000/1024 gives us 2. A total of three segments are used for file record buffers.

3. Similarly, the allocation of the LINKAGE SECTION space is calculated as the sizes of each level 01 or 77 item divided by 1024, and then summed. Thus in our example, the three parameters have sizes of 10, 2000, and 5 bytes, respectively, and are allocated 1, 2, and 1 segment each. The total segment allocation for the LINKAGE SECTION is then 4.

4. The total of these three terms, 4 + 3 + 4, or 11, is thus valid, since it is less than 60.

6.4 Sections

The DATA DIVISION of an MS-COBOL program contains four sections:

FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION
SCREEN SECTION

These sections are described in Sections 6.4.1 through 6.4.4. The SCREEN SECTION for screen data storage and organization is an extension to the full language standard.

6.4.1 FILE SECTION and the File Description (FD) Entry

The FILE SECTION defines the structure of your data files. The File Description (FD) entry and the optional SORT File Description (SD) entry that follow the FILE SECTION header contain entries for every file that has been declared within a SELECT clause in the ENVIRONMENT DIVISION.

The FILE SECTION header appears on its own line and ends with a period (.).

Purpose

Supplies logical and physical descriptions of the files used in the program.

Format

The general format for the FILE SECTION, including FD and SD entry syntax, is:

```
DATA DIVISION.

[ FILE SECTION.

[ FD file-name

     [; BLOCK CONTAINS [ integer-1 TO ] integer-2   {RECORDS    }]
                                                    {CHARACTERS}

     [ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]

      ; LABEL   {RECORD IS  }      {STANDARD}
               {RECORDS ARE}      {OMITTED }

     [; VALUE OF FILE-ID IS   {data-name-1}]
                              {literal-1  }

     [; DATA   {RECORD IS  }   data-name-2 [ , data-name-3 ] ...]
               {RECORDS ARE}

     [; LINAGE IS   {data-name-4}   LINES   [ WITH FOOTING AT   {data-name-5}]
                    {integer-5  }                               {integer-6  }

          [ LINES AT TOP   {data-name-6}]   [ LINES AT BOTTOM   {data-name-7}]]
                           {integer-7  }                        {integer-8  }

     [ ; CODE-SET IS alphabet-name ].

{record-description-entry} ... ] ...

[ SD file-name

     [ ; RECORD CONTAINS [ integer-1 TO ] integer-2 CHARACTERS ]

     [; DATA   {RECORD IS  }   data-name-1 [ , data-name-2 ] ...]
               {RECORDS ARE}

     [; VALUE OF FILE-ID IS   {data-name-1}]
                              {literal-1  }

{record-description-entry} ... ] ...
```

Remarks

FD and SD entries specify the size of the logical and physical records, the presence or absence of label records, the value of implementor-defined label items, names of the data records which make up the file, and the number of lines to be included on a logical printer page. The FD and SD entry ends with a period (.).

The form of the LABEL entry and use of the VALUE OF FILE-ID entry will vary according to the physical destination of your data files.

If you are assigning a data file to PRINTER, the LABEL RECORDS ARE OMITTED entry must appear, and the VALUE OF FILE-ID entry is not permitted in the FD entry.

If you are assigning a data file to DISK, the LABEL RECORDS ARE STANDARD entry must appear, and the VALUE OF FILE-ID clause is required.

The LINAGE and CODE-SET clauses are only relevant in the file description of a Sequential or Line Sequential file.

In the general format of the FILE SECTION, FD entries may be followed by SORT File Description (SD) entries.

The following rules must be observed in the FILE SECTION:

1. The level indicator FD or SD identifies the beginning of a file description. It must be followed by the file-name.
2. The clauses included in the FD or SD entry may be in any order.
3. One or more record description entries must follow the file description entry. Record description entries are discussed in Section 6.2 of this chapter.

Example

```
FILE SECTION.

FD  INVENTORY-MASTER-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "MASTER.DAT".

01  MASTER-RECORD.
    05  MSTR-KEY              PIC X(10).
    05  MSTR-DESCRIPTION      PIC X(25).
    05  MSTR-AMT-ON-HAND      PIC S9(5).
    05  MSTR-WARNING-LEVEL    PIC S9(5).

SD  SORT-FILE
    VALUE OF FILE-ID IS "SORTWORK".

01  SORT-RECORD.
    04  SORT-DATE.
            08  SORT-MONTH  PIC 99.
            08  SORT-DAY    PIC 99.
            08  SORT-YEAR   PIC 99.
    04  SORT-TRANSACTION-CODE   PIC XXX.
    04  SORT-ACCOUNT-NUMBER     PIC 99999.
    04  SORT-REFERENCE  PIC X(9).
    04  SORT-AMOUNT     PIC S9(7)V99
                        SIGN IS LEADING SEPARATE.
```

6.4.2 WORKING-STORAGE SECTION

Purpose

The WORKING-STORAGE SECTION describes the data that are developed and processed internally. These data will not be part of the external data files.

Format

The WORKING-STORAGE SECTION includes the WORKING-STORAGE header, record description, and level 77 entries. The general format is:

```
[ WORKING-STORAGE SECTION.

    [77-level-description-entry]
    [record-description-entry  ]  ... ]
```

Record description entries are described in Section 6.2 of this chapter.

Remarks

Record description and data description entries included in this section may use level-numbers 01 through 49, and 77. Level 77 entries are discussed in Section 6.2.8 of this chapter.

VALUE clauses, which are prohibited in the FILE SECTION, are allowed in the WORKING-STORAGE SECTION.

Example

```
WORKING-STORAGE SECTION.

01  WORK-FIELDS.
    05  MASTER-STATUS       PIC XX      VALUE SPACES.
    05  WARNING-STATUS      PIC XX      VALUE SPACES.
    05  REC-COUNT           PIC S9(5)   VALUE ZERO.
    05  WARNING-COUNT       PIC S9(5)   VALUE ZERO.
    05  END-OF-FILE SW      PIC X       VALUE "N".
        88  END-OF-FILE                 VALUE "Y".
```

6.4.3 LINKAGE SECTION

The LINKAGE SECTION in a program is needed only if the program has been called from another program, and the CALL statement in the calling program contains a USING phrase. The LINKAGE SECTION describes data that are defined in the calling program and are referenced by both the calling and the called programs.

Purpose

To describe data that are referenced by a calling and called program. This section may contain record description entries and level 66, level 77, and level 88 entries.

Format

The LINKAGE SECTION begins with a header, followed by record description entries and level 66, level 77, and level 88 entries. The general format is:

```
[ LINKAGE SECTION.

    [77-level-description-entry ]
    [record-description-entry   ]  ... ]
```

See separate listings in this chapter for details on individual parts of the LINKAGE SECTION.

In the LINKAGE SECTION, the VALUE IS clause may only be used in level 88 entries.

Remarks

No space is allocated in the program for data-items described in the LINKAGE SECTION. Instead, PROCEDURE DIVISION references to these data are resolved by equating the descriptions with data whose addresses are passed to the called program by the CALL statement. Note that for index-items, no such correspondence is established; index-names in the calling and called programs always refer to separate indices.

Data-items that are defined in the LINKAGE SECTION can only be referenced in the PROCEDURE DIVISION of the program if they are specified in the USING phrase of the PROCEDURE DIVISION header or are subordinate to operands in that header.

Because names in the LINKAGE SECTION cannot be qualified, they must be unique within the called program.

See Chapter 8, "Interprogram Communication," for more information on the LINKAGE SECTION.

Example

```
LINKAGE SECTION.

01  SHARED-LIST.
    05  MSTR-DESCRIPTION    PIC X(25).
    05  MSTR-AMT-ON-HAND    PIC S9(5).
```

6.4.4 SCREEN SECTION

Microsoft COBOL is capable of defining screen attributes and having these attributes and other screen definitions displayed in an interactive mode.

This capability comes from using the SCREEN SECTION, which is a Microsoft extension to the DATA DIVISION. This extension takes advantage of Formats 1, 3, and 4 of the ACCEPT statement, and Format 3 of the DISPLAY statement. See Sections 7.6.1 and 7.6.10 for more information about these formats of ACCEPT and DISPLAY, respectively.

Purpose

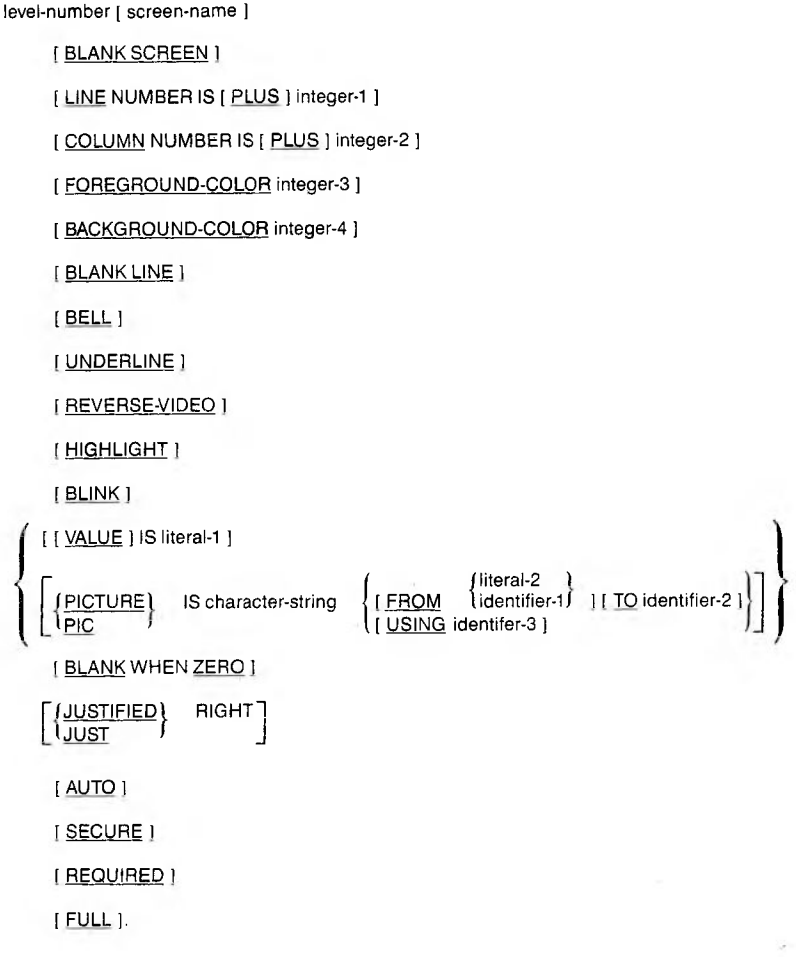
To define terminal format and to describe the data-items entered in the fields on the screen.

Format

The SCREEN SECTION header begins the section, followed by a period (.). Items are entered as group or elementary items, numbered 01 through 49.

Elementary screen items in the SCREEN SECTION define individual display and/or data entry fields. Group items name any group of elementary screen items that are accepted or displayed with a single ACCEPT or DISPLAY statement.

The general format for an elementary screen item is:



The general format for a group screen item is:

```
level-number [ screen-name ]

    [ AUTO ]

    [ SECURE ]

    [ REQUIRED ]

    [ FULL ]
```

Remarks

The clauses used in these formats are discussed in Section 6.5 of this chapter, and may be entered in any order.

If the PICTURE clause is included, either USING or at least one of FROM and TO must be present. The AUTO and SECURE clauses may be used only if the PICTURE clause is also present.

The clauses that are specified with elementary data-items affect data input and display operations when ACCEPT and DISPLAY statements are executed at runtime. These effects are discussed in Sections 6.5.1 through 6.5.33, which describe individual clauses.

With screen items, the following actions are always executed in the order shown below, regardless of the order in which they are specified:

BLANK SCREEN
LINE/COLUMN positioning
BLANK LINE
DISPLAY or ACCEPT data

Example

```
IDENTIFICATION DIVISION.

PROGRAM-ID. DOCTST.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01  WORK-FIELDS.

    05  WS-PASSWORD PIC X(10)
        VALUE "ABCDEFGHIJ".
    05  WS-PART-NO PIC S9(7)
        VALUE 1234567.
    05  WS-DESCRIPTION PIC X(25)
        VALUE "ASDSDKSDDDASABCDEHIJ".
    05  WS-UNIT-COST PIC S99V99
        VALUE 12.34.
    05  WS-QTY-ON-HAND PIC S999
        VALUE 987.

SCREEN SECTION.

01  COLOR-SCREEN.

    02  BLANK SCREEN  FOREGROUND-COLOR 1
            BACKGROUND-COLOR 2.

01  INVENTORY-SCREEN.

    05  LINE 1 COLUMN 1
        VALUE "ENTER PASSWORD ".
    05  COLUMN PLUS 1 PIC X(10) SECURE
        USING WS-PASSWORD.
    05  LINE 2 COLUMN 1
        VALUE "PART NUMBER "
        HIGHLIGHT.
    05  SCR-PART-NO
        LINE 2 COLUMN 13 PIC S9(7)
        USING WS-PART-NO.
    05  LINE 3 COLUMN 1
        VALUE "DESCRIPTION ".
    05  COLUMN PLUS 1 PIC X(25)
        USING WS-DESCRIPTION.
```

```
     05  FOURTH-LINE AUTO.
         10  LINE 4 BLANK LINE.
         10  COLUMN 1 VALUE "UNIT-COST".
         10  COLUMN PLUS 1 PIC S999V99
             BLANK WHEN ZERO
             USING WS-UNIT-COST.
         10  COLUMN PLUS 4
             VALUE "QTY ON HAND".
         10  COLUMN PLUS 1 PIC S999
             BLINK
             USING WS-QTY-ON-HAND.

PROCEDURE DIVISION.

010-MAINLINE.
    DISPLAY COLOR-SCREEN.
    DISPLAY INVENTORY-SCREEN.
    ACCEPT INVENTORY-SCREEN.
    STOP RUN.
```

6.5 Clauses

The remainder of this chapter describes the clauses that may be used within the DATA DIVISION. The clauses are arranged here in alphabetical order. For information about how the clauses are used in the general format, see descriptions of the DATA DIVISON sections in Section 6.4 of this chapter.

The following rules apply to the use of clauses:

1. Clauses may appear in any order except that a REDEFINES clause, if used, must come first.
2. A clause included in a group item applies to all items within that group.
3. If a clause is entered at the group level, it may not be contradicted by a clause in an item that is subordinate to that group.

6.5.1 AUTO Clause

Purpose

Specifies that when a field on a screen has been filled by user input, the cursor automatically skips to the next input field, rather than waiting for a terminating character. The ACCEPT statement is terminated when the last input field is accepted.

Format

The AUTO clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ AUTO ]
```

Remarks

AUTO is effective only when an ACCEPT statement is active during execution of the program.

Example

```
05  LINE 2 COLUMN 13 PIC S9(7)
    TO WS-QTY-ON-HAND
    AUTO.
```

6.5.2 BACKGROUND-COLOR Clause

Purpose

Specifies an integer value that will be interpreted as a background color at runtime.

Format

The BACKGROUND-COLOR clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ BACKGROUND-COLOR integer-4 ]
```

Remarks

BACKGROUND-COLOR sets the color for a single elementary screen item that will be in effect while the screen item is DISPLAYed or ACCEPTed.

When BACKGROUND-COLOR follows a BLANK SCREEN clause, the background color chosen becomes the default color for all following screen items that do not explicitly define colors.

If the BLANK SCREEN clause is not present, the background color is only in effect for the current screen item. If REVERSE-VIDEO is used in the screen item, the values of the foreground and background colors are switched. Integer-4 can be a number from 0 to 15. See the *Microsoft COBOL Compiler User's Guide* for the range of colors represented by integer-4.

If BACKGROUND-COLOR is not supported on a specific terminal, the clause has no effect.

Example

```
05  LINE 2 COLUMN 5
    VALUE "BACKGROUND 2"
    BACKGROUND-COLOR 2.
```

6.5.3 BELL Clause

Purpose

Sounds the terminal's audio alarm.

Format

The BELL clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ BELL ]
```

Remarks

BELL causes the alarm to sound only during an ACCEPT of the field containing the BELL clause.

Example

```
05  LINE 1 COLUMN 1  PIC X(10)
    USING ALARMING-FIELD
    BELL.
```

6.5.4 BLANK LINE Clause

Purpose

Erases the screen from the current cursor position to the end of the current physical line.

Format

The BLANK LINE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ BLANK LINE ]
```

Remarks

The area of the screen in which the specified line appears is cleared. No data are affected.

Example

```
05  LINE 10 BLANK LINE.
```

6.5.5 BLANK SCREEN Clause

Purpose

Erases the entire screen and places the cursor at home position (line 1, column 1).

Format

The BLANK SCREEN clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ BLANK SCREEN ]
```

Remarks

Anything appearing on the screen is erased, but no data are affected.

Example

```
05  BLANK SCREEN.
```

6.5.6 BLANK WHEN ZERO Clause

Purpose

Specifies that a numeric item is displayed as spaces (i.e., is left completely blank) when its value is zero.

Format

The BLANK WHEN ZERO clause may appear in any section within the DATA DIVISION.

The general format is:

```
[ BLANK WHEN ZERO ]
```

Example

```
05  UNIT-COST PIC S999V99
    BLANK WHEN ZERO.
```

6.5.7 BLINK Clause

Purpose

Specifies that an item is to be flashing when displayed on the screen.

Format

The BLINK clause appears as one of several screen attributes in the SCREEN SECTION of the DATA DIVISION. The other available attributes are: UNDERLINE, REVERSE-VIDEO, and HIGHLIGHT.

The general format is:

```
[ BLINK ]
```

Example

```
10  COLUMN PLUS 1 PIC S999
    BLINK
    USING WS-QTY-ON-HAND.
```

6.5.8 BLOCK Clause

Purpose

Specifies the size of the physical records in the file. Because this clause is normally used only for tape files, it is not functional in MS-COBOL. If it is present, however (e.g., if included for transferability), the syntax is checked.

Format

The BLOCK clause appears in an FD entry in the FILE SECTION.

The general format is:

```
[; BLOCK CONTAINS [ integer-1 TO ] integer-2  {RECORDS   }]
                                              {CHARACTERS}
```

Remarks

If the BLOCK clause is specified, the following rules apply:

1. Files assigned to PRINTER must not have a BLOCK clause in the associated FD entry.
2. The size of a physical block should be stated in RECORDS, except when the records are variable in size or exceed the size of a physical block; in these cases the size should be expressed in CHARACTERS.

Example

```
FD  MASTER-INV-FILE
    BLOCK CONTAINS 5 RECORDS.
```

6.5.9 CODE-SET Clause

Purpose

Specifies the character code set used to represent the data on the external media. In MS-COBOL, this clause is used for documentation only.

Format

The CODE-SET clause appears in the FD entry of the FILE SECTION. The code set for MS-COBOL is always ASCII, regardless of the code set specified.

The format is:

```
[ ; CODE-SET IS alphabet-name ].
```

Example

```
FD   INV-RECORD-FILE
     CODE-SET IS ASCII
     .
     .
     .
```

6.5.10 COLUMN Clause

Purpose

Sets the cursor's column position on the screen.

Format

The COLUMN clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ COLUMN NUMBER IS [ PLUS ] integer-2 ]
```

Remarks

The COLUMN and LINE clauses determine the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compile time, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen. When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each item is processed, the current position is adjusted for the size of each definition encountered. By default, therefore, successively defined fields appear end-to-end on the screen.

The current column or line at the start of any elementary screen item data description may be changed with the COLUMN and LINE clauses. If neither clause is specified, the current screen position is not changed. If only COLUMN is specified, the line is not changed. If only LINE is specified, column 1 is assumed.

The COLUMN or LINE clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. When the PLUS phrase is specified, the specified integer is added to the current column

or line, and the result is the column or line at which the current screen item starts. If the integer is not specified, COLUMN/LINE PLUS 1 is assumed.

See also Section 6.5.19, "LINE Clause."

Example

```
05  COLUMN PLUS 4
    VALUE "QUANTITY ON HAND".
```

6.5.11 DATA RECORD(S) Clause

Purpose

Names the records in a file. This clause is used for documentation only.

Format

The DATA RECORD(S) clause appears in FD and SD entries in the FILE SECTION. Note that SD entries are used only with SORT/MERGE files.

The general format is:

```
[ DATA { RECORD IS  } data-name-2 [ , data-name-3 ] ... ]
       { RECORDS ARE }
```

Remarks

Each record in the file is assigned a data-name (e.g., data-name-1, data-name-2, etc.). The records may be of different sizes, formats, etc. The data-names may be listed in any order.

Each data-name must have a corresponding 01 level number record description entry, with the same data-name.

Example

```
FD  RECORD-NAME
    DATA RECORDS ARE TOOLS-1, TOOLS-2
      .
      .
      .
01  TOOLS-1.
    05  PART-NO PIC 9(8).
    05  DESCRIPTION PIC X(25).
    05  QTY PIC 999.

01  TOOLS-2.
    05  PART-NO PIC 9(8).
    05  COST PIC 9(9)V99.
```

6.5.12 FOREGROUND-COLOR Clause

Purpose

Specifies an integer value that will be interpreted as a foreground color at runtime.

Format

The FOREGROUND-COLOR clause appears as part of the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ FOREGROUND-COLOR integer-3 ]
```

Remarks

FOREGROUND-COLOR sets the color for a single elementary screen item that will be in effect while the screen item is DISPLAYed or ACCEPTed.

When FOREGROUND-COLOR follows a BLANK SCREEN clause, the foreground color chosen becomes the default color for all following screen items that do not explicitly define colors.

If the BLANK SCREEN clause is not present, the foreground color is only in effect for the current screen item. If REVERSE-VIDEO is used in the screen item, the values of the foreground and background colors are switched. Integer-3 can be a number from 0 to 15. See the *Microsoft COBOL Compiler User's Guide* for the range of colors represented by integer-3 for your terminal or computer.

If FOREGROUND-COLOR is not supported on a specific computer or terminal, the clause has no effect.

Example

```
05  BLANK SCREEN FOREGROUND-COLOR 4.
```

6.5.13 FROM/TO/USING Clause

Purpose

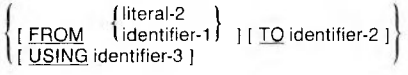
When a data-item is displayed on a screen, FROM or USING moves the contents of the data-item or a literal from storage to a temporary item that is defined by the PICTURE clause. This value is then displayed on the screen.

When an item is accepted, TO or USING implicitly moves the contents of the item to the data-item named in the TO or USING clause.

Format

The FROM/TO/USING clause appears in the SCREEN SECTION of the DATA DIVISION, and is part of the PICTURE clause for a data-item associated with a screen.

The general format is:



Remarks

Identifiers may be qualified but not subscripted.

Note that the FROM and TO clauses are used together; USING, in effect, combines the two.

Examples

```
05  LINE 1 PIC S9(5)
    USING WS-PART-NO.

05  SCR-DESC PIC X(25)
    FROM LS-DESCRIPTION.

05  COLUMN PLUS 1 PIC X(10)
    TO FILE-IDENT.
```

The first example references data in the WORKING-STORAGE SECTION; the second example references data in the LINKAGE SECTION; and the third example references data in the FILE SECTION.

6.5.14 FULL Clause

Purpose

When a data-item is accepted from the screen, FULL causes any terminator characters to be ignored until the field is completely filled.

Format

The FULL clause is used in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ FULL ]
```

Example

```
05  LINE 3 PIC X(5)
    TO WS-IDENT-NO
    FULL.
```

6.5.15 HIGHLIGHT Clause

Purpose

Specifies that an item is shown in a highlighted mode when displayed on the screen.

Format

The HIGHLIGHT clause appears as one of several screen attributes in the SCREEN SECTION of the DATA DIVISION. The other attributes are: BLINK, UNDERLINE, and REVERSE-VIDEO.

The general format is:

```
[ HIGHLIGHT ]
```

Example

```
05  LINE 2 VALUE "ENTER UNIT COST"
    HIGHLIGHT.
```

6.5.16 JUSTIFIED Clause

Purpose

Specifies right-to-left alignment when the field is the receiving field for a MOVE statement.

Format

The JUSTIFIED clause may appear in any section of the DATA DIVISION. The abbreviated form, JUST, is allowed.

The general format is:

```
[{JUSTIFIED} RIGHT]
 {JUST     }
```

Remarks

The JUSTIFIED clause applies only to unedited alphanumeric items. It can be used only for elementary items.

When the receiving field is longer than the source field, the data are aligned right-to-left, with space fill on the left. When the receiving field is shorter than the source field, truncation occurs from the left.

Example

```
05  ALPHA-ITEM PIC X(20)
    JUSTIFIED RIGHT.
```

6.5.17 LABEL RECORD(S) Clause

Purpose

Indicates whether a file contains labels.

Format

The LABEL RECORD(S) clause appears in the FD entry of the FILE SECTION.

The general format is:

```
:LABEL  {RECORD IS  }   {STANDARD}
        {RECORDS ARE}   {OMITTED  }
```

OMITTED specifies that no labels exist for the file. OMITTED must be specified for files assigned to PRINTER.

STANDARD specifies that labels exist for the file and that they conform to system specifications. STANDARD must be specified for files assigned to DISK.

Remarks

This clause is required in every FD entry.

Example

```
FD  INVENTORY-WARNING-FILE
    LABEL RECORDS ARE STANDARD.
```

6.5.18 LINAGE Clause

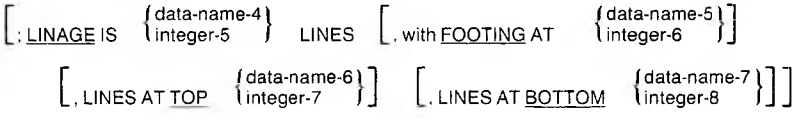
Purpose

Specifies the total number of lines assigned to a printed page, the number of lines allotted for top and bottom margins, and the line number at which the footing (information printed at the bottom of the page) begins.

Format

The LINAGE clause appears in the FD entry of the FILE SECTION for a file assigned to PRINTER.

The general format is:



Remarks

All data-names refer to unsigned numeric integer data-items. The operands of the LINAGE and FOOTING phrases must have values greater than zero. The values of the operands of the LINES AT TOP and LINES AT BOTTOM phrases may be zero. The operand in the FOOTING phrase must have values not greater than that in the LINAGE phrase operand.

The total page size is the sum of the values in each phrase except for FOOTING. If TOP or BOTTOM margins are not specified, their size is assumed to be zero. The footing area is the part of the page between the line indicated by the FOOTING value and the last line of the page.

The values in each phrase at the time the file is opened (by the execution of an OPEN OUTPUT statement) specify the number of lines in each of the sections of the first logical page. Whenever a WRITE statement with the ADVANCING PAGE phrase is executed or a "page overflow" condition occurs (see Section

7.6.39, "WRITE Statement"), the values in the phrases will be used for the next logical page.

A LINAGE-COUNTER is automatically created by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the printer is positioned within the current page. LINAGE-COUNTER may be referenced but may not be modified by PROCEDURE DIVISION statements. It is automatically modified during execution of a WRITE statement, according to the following rules:

1. When the ADVANCING PAGE phrase of the WRITE statement is specified or a "page overflow" condition occurs (see Section 7.6.39, "WRITE Statement"), LINAGE-COUNTER is reset to one.
2. When the ADVANCING {identifier | integer} phrase is specified, LINAGE-COUNTER is incremented by the ADVANCING value.
3. When the ADVANCING phrase is not specified, LINAGE-COUNTER is incremented by one.

Example

```
FD  INVENTORY-REPORT-FILE
    LABEL RECORDS ARE OMITTED
    LINAGE IS 56 LINES
    LINES AT TOP 3
    LINES AT BOTTOM 5.
```

6.5.19 LINE Clause

Purpose

Sets the line position of the cursor on the screen.

Format

The LINE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ LINE NUMBER IS [ PLUS ] integer-1 ]
```

Remarks

The COLUMN and LINE clauses determine the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compile time, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen. When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each item is processed, the current position is adjusted for the size of each definition encountered. By default, therefore, successively defined fields appear end-to-end on the screen.

The current column or line at the start of any elementary screen item data description may be changed with the COLUMN and LINE clauses. If neither clause is specified, the current screen position is not changed. If only COLUMN is specified, the line is not changed. If only LINE is specified, column 1 is assumed.

The COLUMN or LINE clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. When the PLUS phrase is specified, the specified integer is added to the current column or line, and the result is the column or line at which the

current screen item starts. If the integer is not specified, COLUMN/LINE PLUS 1 is assumed.

Example

```
05  LINE 1 PIC 999
    USING WS-QUANTITY.
```

6.5.20 OCCURS Clause

Purpose

Specifies the number of times that a data-item is repeated in a record, for example within the associated level 01 group. The OCCURS clause allows for a variable number of repetitions of a data-item.

Format

The general format is:

```
[; OCCURS  { integer-1 TO integer-2 TIMES DEPENDING ON data-name-3 }
           { integer-2 TIMES                                       }

    [ { ASCENDING  } KEY IS data-name-4 [ , data-name-5 ] ... ] ...
      { DESCENDING }

    [ INDEXED BY index-name-1 [ , index-name-2 ] ... ] ]
```

The OCCURS clause is used to define the number of occurrences of table elements and the subscript or index-name that will apply during table handling.

The DEPENDING ON phrase specifies that the subject of the entry has a variable number of occurrences, the number of which must fall within the range defined by integer-1 and integer-2.

The KEY and INDEXED phrases specify key-names that can be used in SEARCH statements to refer to the items within a table. In the KEY phrase, ASCENDING and DESCENDING specify that the items in the table are arranged in ascending or descending order according to their values.

See Chapter 9, "Table Handling by the Indexing Method," for more information about the KEY and INDEXED phrases.

Remarks

The OCCURS clause defines related sets of repeated data, such as tables, lists, and arrays. It specifies the number of times that a data-item with the same format is repeated in the record. The entire data description entry applies to each repetition of the entry.

A data-item may have at most three subscripts (e.g., TABLE-VAL (2, IIX, WS-LEVEL)). Therefore, up to three OCCURS clauses may be nested in a single data record.

The OCCURS clause may be used in the FILE, WORKING-STORAGE, and LINKAGE SECTIONS. The OCCURS clause may not be used in a level-number 01, 66, 77, or 88 entry, or in a data description that is subordinate to another data description that itself contains an OCCURS clause modified by the DEPENDING ON phrase.

When the OCCURS clause is used in an entry, the data-name associated with the OCCURS clause must be subscripted or indexed whenever it appears as an operand in the PROCEDURE DIVISION. If this data-name is the name of a group item, all data-names belonging to the group must be subscripted or indexed whenever they are used.

Subscripting enables the user to refer to a table or list of data-items that have not been assigned individual data-names. This is the case for items that have been specified by an OCCURS clause; therefore, any item that contains an OCCURS clause or belongs to a group containing an OCCURS clause must be subscripted or indexed whenever it is used. The one exception is in a SEARCH statement, where a table-name must be used without subscripts. See Section 9.2, "Subscripting," for more information.

A subscript is a positive, nonzero integer whose value indicates which element is selected within a table or list. The subscript may be either a literal or a data-name whose value is an integer. A subscript must be defined as a decimal or binary item (USAGE IS DISPLAY for decimal format or either COMPUTATIONAL-0 or COMPUTATIONAL-4 for the binary format; the binary format is recommended for efficiency).

A subscript is always enclosed by parentheses. In the general format, it is given after the terminal space of the name of the element. Multiple subscripts are separated by a comma and a space (e.g., ITEM (3, 4)).

A data-name may not be subscripted if it is being used for:

1. a subscript
2. the defining name of a data description entry
3. data-name-2 in a REDEFINES clause
4. a qualifier

Examples

```
01   ARRAY.
     03 TABLE-VAL OCCURS 3 TIMES PIC 9(4).
```

In this example, storage would be allocated as follows:

```
TABLE-VAL (1)
TABLE-VAL (2)
TABLE-VAL (3)
```

These three occurrences make up the ARRAY, which consists of 12 characters (each TABLE-VAL has 4 digits).

```
01   DEPENDING-ARRAY.
     03  TABLE OCCURS 1 TO 100 TIMES
           DEPENDING ON TSIZE
           INDEXED BY DEP-IND.

         05  TABLE-ENTRY   PIC X(4).
```

In this example, the value of TSIZE determines the number of entries that DEPENDING-ARRAY can hold at a given time.

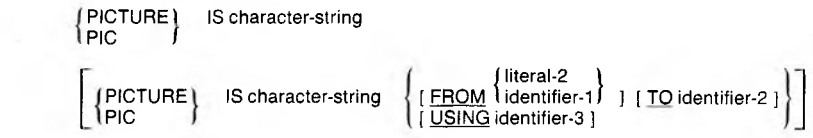
6.5.21 PICTURE Clause

Purpose

Describes the contents of every elementary data-item, except an index-data-item. May also describe editing features of the item.

Format

The general format is:



The abbreviation PIC is allowed.

The second format is used only in the SCREEN SECTION, where the PICTURE clause must be followed by the USING clause, or one or both of the FROM and TO clauses.

Character-strings are discussed in the remarks which follow.

See Section 6.5.13 for discussion of the FROM/TO/USING clause.

Remarks

The character-string specification differs for alphanumeric, alphanumeric-edited, numeric, and numeric-edited data-items. These differences are described in the following paragraphs.

Alphanumeric and Alphanumeric-Edited Items

The PICTURE clause of an alphanumeric item may combine characters X, A, and 9. It may also contain the editing characters B, 0, and /, in which case it is considered to be an alphanumeric-edited item.

An X indicates that the character position may contain any character from the computer's character set. A PICTURE clause that contains at least one of these combinations:

A and 9

X and 9

X and A

in any order, is considered as if every X, A, or 9 character were X.

The characters B, 0, and / may be used to insert blanks, zeros, or slashes in the item.

If the string contains only A's and B's, it is considered alphabetic; if it has only 9's, it is numeric. The NUMERIC and ALPHABETIC class tests may be used to determine whether an alphanumeric data-item is alphabetic or numeric.

Numeric Items

The PICTURE clause of a numeric item may combine the following characters:

9 Indicates that the actual or conceptual digit position contains a numeric character. The maximum number of 9's in a PICTURE clause is 18.

V Indicates the position of an assumed decimal point. This character is optional. Since a numeric item cannot contain an actual decimal point, the assumed decimal point provides the compiler with information about the scaling alignment of items involved in computations. Storage is never reserved for the character V. Only one V is permitted in any single PICTURE clause. V is redundant if it is the rightmost character.

S Indicates that the item has an operational sign. This character is optional. It must be the first character of the PICTURE clause. See also Section 6.5.27, "SIGN Clause."

P Indicates an assumed decimal scaling position. This character is optional. It specifies the location of an assumed decimal point when the point is not within the number that appears in the data-item.

The scaling position character P is not counted in the size of the data-item, and memory is not reserved for it. However, scaling position characters are counted in determining the maximum number of digit positions (18) in numeric-edited items or in items that appear as operands in arithmetic statements.

If the clause contains more than one P, the P's must be continuous. The character P may appear only to the left or right of the other characters in the string, except that it may appear to the left of a leftmost string of P's. P implies an assumed decimal point to the left of the P's if the P's are leftmost, and to the right of the P's if the P's are rightmost. Therefore, the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within a PICTURE clause that contains P's.

Numeric-Edited Items

A numeric-edited item is a data-item that can be used as an "edited" receiving field for a numeric value. The editing characters that may be combined to describe a numeric-edited item are:

9 V . Z CR DB , $ + * B 0 - P /

The characters 9, V, and P have the same meaning as for a numeric item.

Editing for numeric-edited items may be insertion editing or suppression and replacement editing. The editing characters that are used for these editing tasks are described in the paragraphs that follow.

Insertion editing. The four types of insertion editing are: simple insertion, special insertion, fixed insertion, and floating insertion.

1. simple insertion

   , The comma specifies insertion of a comma between digits. Each comma is counted in the size of the data-item, but does not represent a digit position. The comma may also appear in conjunction with a floating string, as described below. It must not be the last character in the PICTURE character-string.

   B Each appearance of B in a PICTURE clause represents a blank in the final edited value.

   / Each slash in a PICTURE clause represents a slash in the final edited value.

   0 Each appearance of zero in a PICTURE clause represents a position in the final edited value where the digit zero will appear.

2. special insertion

   . The decimal point specifies that an actual decimal point is to be inserted in the indicated position and that the source item is to be aligned accordingly. Numeric character positions to the right of an actual decimal point in a PICTURE clause must consist of characters of one type. The decimal point must not be the last character in the PICTURE character-string. The decimal point and P may not be used in the same PICTURE clause.

3. fixed insertion

   + − The plus sign (+) or minus sign (−) may appear in a PICTURE clause either singly or in a floating string. As a fixed-sign character, the + or − must appear as the last symbol in the PICTURE clause.

   The plus sign indicates that the sign of the item is indicated by either a plus or minus placed in the character position, depending on the algebraic sign of the numeric value placed in the output field. The minus sign indicates that blank or minus is placed in the character position, depending on whether the algebraic sign of the numeric value placed in the output field is positive or negative, respectively.

   CR DB CR and DB are credit and debit symbols, respectively. They may appear only as the rightmost characters in a PICTURE clause. These symbols occupy two character positions. They indicate that the specified symbol is to appear in the indicated positions if the value of a source item is negative. If the value is positive or zero, spaces will appear instead. CR and DB and + and − are mutually exclusive.

4. floating insertion

   A floating string is a leading, continuous series of either dollar signs, plus signs, or minus signs; or a string composed of one such character interrupted by one or more commas and/or decimal points. For example:

```
$$,$$$,$$$
++++
--,---,---
+(8).++
$$,$$$.$$
```

   A floating string containing (N + 1) occurrences of ($) or (+) or (−) defines N digit positions. When a numeric value is placed in a numeric-edited item, the numeric-edited item will have one actual $ or + or −

immediately to the left of the most significant nonzero digit, in one of the positions indicated by $ or + or − in the PICTURE clause. Blanks are placed in all character positions to the left of the float character.

If the most significant digit appears in a position to the right of positions defined by the floating string, the numeric-edited item will contain $ or + or − in the rightmost position of the floating string, and nonsignificant zeros may follow. When a floating string contains an actual or implied decimal point, all digit positions to the right of the decimal point are treated as if they contained 9's.

When a comma (,) appears to the right of a floating string, the float character disregards the comma so that it may be as close to the leading digit as possible.

In the following examples, "b" represents a blank in the developed items.

PICTURE Clause	Numeric Value	Printed Output
$$$999	14	bb$014
--,---,999	-456	bbbbbb-456
$$$$$$	14	bbb$14

A floating string need not constitute the entire PICTURE clause of a numeric-edited item.

Suppression and replacement editing.

Z
*

The characters Z and * are suppression and replacement characters. Each one represents a digit position. During execution, leading zeros to be placed in positions defined by Z or * are suppressed, becoming blank or *, respectively. Zero suppression ends when a decimal point (. or V) or a non-zero digit is encountered. All digit positions to be modified must be the same (either Z or *), and must be contiguous starting from the left. Z or * may appear to the right of an actual decimal point only if all digit positions are the same.

Commas will not be inserted into numeric-edited items where the zero to the left of the comma has been suppressed. Instead, the comma will be replaced by either a blank or an asterisk.

Other rules for the PICTURE clause of a numeric-edited item are:

1. Only one type of floating string may be used in the item.
2. The item must have at least one digit position character.
3. The appearance of a floating sign string or fixed plus or minus character precludes the appearance of any other of the sign control insertion characters, namely, +, −, CR, and DB.
4. The characters from the immediate right of a decimal point to the end of the PICTURE clause (excluding the fixed insertion characters +, −, CR, and DB), are subject to the following restrictions:
   a. Only one type of digit position character may appear. That is, only one of Z, *, or 9, and the floating-string digit position characters $, +, or − may be used.
   b. If one of the numeric character positions to the right of a decimal point is represented by + or − or $ or Z, then all the numeric character positions in the PICTURE clause must be represented by the same character.
5. The PICTURE character 9 can never appear to the left of a floating string or replacement character.

Additional notes on the PICTURE clause:

1. A maximum of 30 character positions is allowed in a PICTURE character string. For example, PICTURE X(89) consists of the five PICTURE characters:

   X (8 9)

2. A PICTURE clause must contain at least one of the characters A, Z, *, X, 9 or at least two consecutive appearances of the symbols +, −, or $.
3. The characters ., S, V, CR, and DB can appear only once in a PICTURE clause.
4. When the DECIMAL-POINT IS COMMA clause is specified in the ENVIRONMENT DIVISION of the program, the explanations for period and comma apply to comma and period, respectively.
5. A PICTURE clause is used only with elementary items, not with group items.

The following examples illustrate how data are represented by the PICTURE clause. "Data Value" shows contents in storage; and "Edited Data" shows the value that is reported.

Source Area		Receiving Area	
PICTURE	**Data Value**	**PICTURE**	**Edited Data**
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00123	$$$,$$9.99	$123.00
9(5)	00000	$$$,$$9.99	$0.00
9(4)V9	12345	$$$,$$9.99	$1,234.50
V9(5)	12345	$$$,$$9.99	$0.12
S9(5)	00123	--------.99	123.00
S9(5)	-00001	--------.99	-1.00
S9(5)	00123	+++++++.99	+123.00
S9(5)	00001	--------.99	1.00
9(5)	00123	+++++++.99	+123.00
9(5)	00123	--------.99	123.00
S9(5)	12345	*******.99CR	**12345.00
S999V99	02345	ZZZ.ZZ	23.45
S999V99	00004	ZZZ.ZZ	.04

6.5.22 RECORD Clause

Purpose

Specifies the number of characters each record in the file contains. It is used to check that the length of the records defined within the FD and SD is within the range of lengths specified by this clause. The size of each data record is actually defined by the data description entries that make up the record (level 01) declaration.

Format

The RECORD clause appears in the FD (and SD) entry in the FILE SECTION.

The general format is:

```
[ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]
```

Integer-4 should be the size of the largest record in the file declaration. If the records are variable in size, integer-3 must be specified and equal to the size of the smallest record. The sizes are given as character positions required to store the logical records.

Remarks

This clause is always optional.

Examples

```
FD INV-MSTR-FILE
   RECORD CONTAINS 80 CHARACTERS.

FD VAR-MSTR-FILE
   RECORD CONTAINS 20 TO 200 CHARACTERS.
```

6.5.23 REDEFINES Clause

Purpose

Specifies that a storage area is to contain different data-items, or provides an alternative grouping or description of the same data.

Format

The REDEFINES clause is optional. If present, it must be the first clause in the data description or record description entry.

The general format is:

```
[ ; REDEFINES data-name-2 ]
```

The data description entry for data-name-2 should not contain a REDEFINES clause or an OCCURS clause.

Warning

We strongly advise against redefining COMP-0 or COMP-4 data-items to refer to parts of such a data-item. The order in which bytes are stored may vary among implementations, or may change in later versions of Microsoft COBOL. Such a practice may limit the portability of your software.

Remarks

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items that share the same storage area due to redefinition, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C.

For purposes of discussing redefinition, data-name-1 is termed the subject, and data-name-2 is called the object. The levels of the subject and object are denoted by s and t, respectively. The following rules must be obeyed in order to establish a proper redefinition:

1. Level s must equal level t, but must not equal 66 or 88.
2. The object must be contained in the same record (01 group level item), unless s = t = 01.
3. The REDEFINES clause may not be used in level 01 entries in the FILE SECTION, because multiple 01 level items in the FILE SECTION are implicitly redefined.
4. Prior to definition of the subject and subsequent to definition of the object, there can be no level numbers that are numerically less than s.
5. The length of data-name-1, multiplied by the number of occurrences of data-name-1, may not exceed the length of data-name-2, unless the level of data-name-1 is 01.
6. Data-name-1 and entries subordinate to data-name-1 must not contain any VALUE clauses, except in level 88.

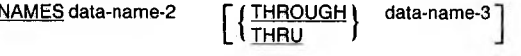
6.5.24 RENAMES Clause

Purpose

Provides for alternative grouping of data-items subordinate to the same 01 level data-item.

Format

The general format is:



Remarks

The following clauses may be used to modify the RENAMES clause:

THROUGH | THRU

The level 66 entry and RENAMES clause must immediately follow the last data-item entry for the record that it modifies. Data-name-1 cannot be used as a qualifier. Data-name-3 must follow data-name-2 in the record description. A level 66 entry cannot rename a level 01, 66, 77, or 88 entry. Data-name-1 is described with the level 66 entry in Section 6.2, "Record Description Entry."

The range of data renamed consists of all data from the beginning of data-name-2 to the end of data-name-3, or the end of data-name-2 if data-name-3 is not specified.

Neither data-name-2 nor data-name-3 may have an OCCURS clause in its description nor be subordinate to an item that contains an OCCURS clause. None of the items within the renamed range may contain the OCCURS clause with the DEPENDING ON option.

If data-name-3 is specified, or data-name-2 is a group item, then data-name-1 is a group item. Otherwise, data-name-1 is an elementary item with the same characteristics as data-name-2.

Example

```
05  A-DATA-GROUP.
    10  ITEM1          PIC X(10).
    10  ITEM2          PIC 9(5).
    10  ITEM3          PIC 9(3).
    10  ITEM4          PIC X(7).

    66  ANOTHER RENAMES A-DATA-GROUP THRU ITEM3.
```

6.5.25 REQUIRED Clause

Purpose

When a data-item is accepted from the screen, REQUIRED causes terminator characters to be ignored until at least one non-terminator character is entered into a field.

The REQUIRED clause appears in the SCREEN SECTION of the DATA DIVISION.

Format

The general format is:

```
[ REQUIRED ]
```

Example

```
05  LINE 3 PIC X(5)
    TO WS-IDENT-NO
    REQUIRED.
```

6.5.26 SECURE Clause

Purpose

Suppresses the echo of characters input at the terminal. Instead, an asterisk is displayed for each data character accepted.

Format

The SECURE clause appears in the SCREEN SECTION of the DATA DIVISION.

The general format is:

```
[ SECURE ]
```

Remarks

The SECURE clause is always optional.

Example

```
05  SCREEN-NAME PIC S9(5)
    USING WS-NAME
    SECURE.
```

6.5.27 SIGN Clause

Purpose

Specifies that an operational sign be included as part of an external decimal item; also specifies one of four possible formats for placement of the sign.

Format

The SIGN clause appears in the data description entry for an external decimal item (USAGE IS DISPLAY).

The general format is:

```
[ [ SIGN IS ]  { LEADING  }  [ SEPARATE CHARACTER ] ]
               { TRAILING }
```

where the possible forms of the clause are:

SIGN Clause	Sign Representation
TRAILING	Embedded in rightmost byte
LEADING	Embedded in leftmost byte
TRAILING SEPARATE	Stored in separate rightmost byte
LEADING SEPARATE	Stored in separate leftmost byte

The SEPARATE CHARACTER phrase increases the size of the data-item by one character.

Remarks

The following rules apply to the SIGN clause:

1. When an operational sign is specified, the PICTURE must begin with S. If no S is used, the item is not signed (and is capable of storing only absolute values), and the SIGN clause is prohibited. When S appears at the front of a PICTURE but no SIGN clause is included in an item's description, the default case, SIGN IS TRAILING, is assumed.
2. The SIGN clause may be written at the group level. In this case, the clause specifies the sign's format on any signed subordinate external decimal item.
3. The entries to which the SIGN clause applies must be implicitly or explicitly described as USAGE IS DISPLAY.
4. When the CODE-SET clause is specified for a file, all signed numeric data for that file must be described with the SIGN IS SEPARATE clause.

6.5.28 SYNCHRONIZED Clause

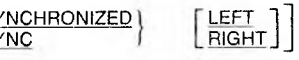
Purpose

The SYNCHRONIZED clause was designed in order to efficiently allocate memory space for data. It specifies the alignment of an item on the computer's natural memory boundaries. However, in MS-COBOL, the SYNCHRONIZED clause is used for documention only.

Format

The SYNCHRONIZED clause is used in the standard data description entry.

The general format is:



Remarks

The SYNCHRONIZED clause may be used only with elementary data-items.

Although this clause is for commmentary only, the compiler does check the syntax.

Example

```
05  RECORD-ITEM PIC X(10)
    SYNCHRONIZED RIGHT.
```

6.5.29 TO Clause

See Section 6.5.13, "FROM/TO/USING Clause."

6.5.30 USAGE Clause

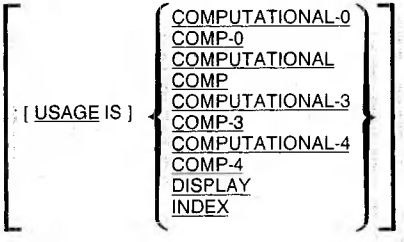
Purpose

Specifies the form in which numeric data are represented.

Format

The USAGE clause appears in data description or record description entries in the FILE, WORKING-STORAGE, or LINKAGE SECTIONS.

The general format is:



COMP is an accepted abbreviation for COMPUTATIONAL.

A COMPUTATIONAL item is capable of representing a value to be used in computations. It must be numeric. If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group item itself is not COMPUTATIONAL and cannot be used in computations.

COMPUTATIONAL-3, which may be abbreviated COMP-3, defines a packed binary-coded (internal decimal) field. COMPUTATIONAL-0 (abbreviated COMP-0) defines a two-byte binary integer. COMPUTATIONAL-4 (COMP-4) defines a four-byte binary integer.

Warning

We strongly advise against redefining COMP-0 or COMP-4 data-items to refer to parts of such a data-item. The order in which bytes are stored may vary among implementations, or may change in later versions of Microsoft COBOL. Such a practice may limit the portability of your software.

The USAGE IS DISPLAY clause indicates that the data are in standard ASCII data format.

USAGE IS INDEX indicates that the data-item will be used as an index-data-item (see Chapter 9, "Table Handling by the Indexing Method").

USAGE IS INDEX defines the data-item to be a binary item, in the same format as a COMPUTATIONAL-0 data-item. If USAGE IS INDEX is used, no PICTURE clause can be used.

Remarks

If a USAGE clause is given at a group level, it applies to each elementary item in the group. The USAGE clause for an elementary item must not contradict the USAGE clause of a group to which the item belongs.

The USAGE clause may be written at any level. If USAGE is not specified, the item is assumed to be USAGE IS DISPLAY.

The USAGE IS COMPUTATIONAL-3 clause is required in the data description for an internal decimal number.

The USAGE IS COMPUTATIONAL-0 or COMPUTATIONAL-4 clause is advised for subscripts.

Example

```
05  TOTAL-AMT-SALES PIC S9(5)V99
    USAGE IS COMP-3.
```

6.5.31 USING Clause

See Section 6.5.13, "FROM/TO/USING Clause."

6.5.32 VALUE IS Clause

Purpose

Specifies the initial value of data-items or conditions.

Format

In MS-COBOL, the VALUE IS clause appears only in the WORKING-STORAGE and SCREEN SECTIONS or in level 88 conditions. The format for a standard data description entry is:

```
[ ; VALUE IS literal ]
```

The format for a level 88 condition-name is:

```
88 condition-name;  { VALUE IS  }  literal-1  [ { THROUGH } literal-2 ]
                    { VALUES ARE}             [ { THRU    }           ]

                    [ , literal-3  [ { THROUGH } literal-4 ] ]  ...
                                   [ { THRU    }           ]
```

THROUGH and THRU are equivalent.

Note that the VALUE IS clause is required for level 88 conditions.

Remarks

The VALUE IS clause must not be written in a data description entry that also has an OCCURS or REDEFINES clause, or in an entry that is subordinate to an entry containing an OCCURS or REDEFINES clause. Furthermore, it cannot be used in the FILE or LINKAGE SECTIONS, except in level 88 condition descriptions.

The size of the literal given in a VALUE IS clause must be less than or equal to the size of the item as given in the PICTURE clause. The positioning of the literal within a data area is the same as would result from specifying a MOVE of the literal to the data area, except that editing characters in the PICTURE have no effect on the initialization, nor do BLANK WHEN ZERO or JUSTIFIED clauses.

The type of literal written in a VALUE IS clause depends on the type of data-item, as described in Chapter 2, "Language Elements." For edited items, values must be specified as non-numeric literals, and must be presented in edited form.

In the SCREEN SECTION, the literal must be a non-numeric (quoted) literal and cannot be a figurative constant; in other sections, the literal can be a numeric, non-numeric, or figurative constant.

When an initial value is not specified, no assumption should be made regarding the initial contents of an item in WORKING-STORAGE.

The VALUE IS clause may be specified at the group level, in the form of a correctly sized non-numeric literal, or as a figurative constant. In these cases the VALUE IS clause cannot be stated at the subordinate levels within the group. However, the VALUE IS clause should not be written for a group containing items with descriptions that include JUSTIFIED, SYNCHRONIZED, and USAGE clauses (other than USAGE IS DISPLAY).

See Section 6.5.33 for a description of the VALUE OF FILE-ID clause, which provides information for the label records associated with a disk file.

Examples

```
05 QTY PIC 99 VALUE IS 24.
   88 ON-HAND-QTY VALUE IS 1 THRU 3.
```

6.5.33 VALUE OF FILE-ID Clause

Purpose

Provides the operating system with information for the label records associated with a file that is assigned to DISK.

Format

The VALUE OF FILE-ID clause appears in any FD or SD entry for a disk-assigned file. The VALUE OF FILE-ID clause contains a FILE-ID expressed as a data-name or non-numeric literal.

The general format is:

```
: VALUE OF FILE-ID IS {data-name-1}
                      {literal-1   }
```

Remarks

If a file is assigned to PRINTER, it is unlabelled and the VALUE OF FILE-ID clause must not be included in the associated FD or SD. If a file is assigned to DISK, it is necessary to include both LABEL RECORDS STANDARD and VALUE OF FILE-ID clauses in the associated FD or SD. If a data-name is specified, it may contain as many characters as desired. See your *Microsoft COBOL Compiler User's Guide* for FILE-ID formats for specific operating systems.

Examples

```
(MS-DOS) VALUE OF FILE-ID IS "A:MASTER.ASM".
(Xenix)  VALUE OF FILE-ID IS "/usr/bobz/xout.lst".
```

Chapter 7
PROCEDURE DIVISION

This chapter describes the statements used in the PROCEDURE DIVISION of a Microsoft COBOL program. Descriptions of individual statements that provide the structure of the source text are arranged alphabetically in Section 7.6, "PROCEDURE DIVISION Statements."

The PROCEDURE DIVISION of a COBOL program contains the logic necessary for solving a data processing problem. The instructions for file I-O, decision-making, and arithmetic operations are coded into this division of the source program.

7.1 PROCEDURE DIVISION Header and General Format

Purpose

To initiate the data processing procedures required for solving a given problem.

Format

The general format for the PROCEDURE DIVISION is:

```
PROCEDURE DIVISION      [ { USING    } data-name-1[ , data-name-2 ]... ]
                          { CHAINING }

[ DECLARATIVES.

{ section-name SECTION [ segment-number ]. USE statement.

[ paragraph-name. [ sentence] ... ] ... } ...

 END DECLARATIVES. ]

{ section-name SECTION [ segment-number ].

[ paragraph-name. [ sentence ] ... ] ... } ...
```

See Chapter 8, "Interprogram Communication," for the general format of the PROCEDURE DIVISION for programs that are invoked by the CALL or CHAIN statements and options that require use of the USING or CHAINING phrases.

Remarks

The PROCEDURE DIVISION may be subdivided in three possible ways:

1. It may consist only of paragraphs.
2. It may consist of a number of paragraphs followed by a number of sections (each section subdivided into one or more paragraphs).
3. It may consist of a DECLARATIVES Region and a series of sections (each section subdivided into one or more paragraphs).

   Using the DECLARATIVES Region as a subdivision of the PROCEDURE DIVISION is optional; it provides a means of designating procedures to be invoked in the event of an I-O error. See Chapter 14, "DECLARATIVES Region and USE Statement," for details on the DECLARATIVES Region syntax.

Example

```
PROCEDURE DIVISION.

P000-MAINLINE.
    OPEN INPUT INVENTORY-MASTER-FILE,
         OUTPUT INVENTORY-WARNING-FILE,
         INVENTORY-REPORT-FILE.

    WRITE REPORT-RECORD FROM PR-HEADER
         AFTER ADVANCING PAGE.
    PERFORM P100-WRITE-REPORT
         UNTIL END-OF-FILE.

    MOVE REC-COUNT TO PR-REC-COUNT.
    MOVE WARNING-COUNT TO PR-WARNING-COUNT.
    WRITE REPORT-RECORD
         FROM PR-TOTAL-RECORD
         AFTER ADVANCING 2 LINES.

    CLOSE INVENTORY-MASTER-FILE,
         INVENTORY-WARNING-FILE,
         INVENTORY-REPORT-FILE.

    STOP RUN.
```

```
P100-WRITE-REPORT.
    READ INVENTORY-MASTER-FILE
         AT END MOVE "Y" TO END-OF-FILE-SW.
    IF NOT END-OF-FILE
         PERFORM P200-PROCESS-RECORD.

P200-PROCESS-RECORD.
    MOVE MSTR-KEY TO PR-KEY.
    MOVE MSTR-DESCRIPTION
         TO PR-DESCRIPTION.
    MOVE MSTR-AMT-ON-HAND
         TO PR-AMT-ON-HAND.
    MOVE MSTR-WARNING-LEVEL
         TO PR-WARNING-LEVEL.
    PERFORM P300-WRITE-LINE.

    IF MSTR-AMT-ON-HAND <
       MSTR-WARNING-LEVEL
         MOVE MASTER-RECORD
              TO WARNING-RECORD
         ADD 1 TO WARNING-COUNT
         WRITE WARNING-RECORD.

P300-WRITE-LINE.
    WRITE REPORT-RECORD
         FROM PR-REPORT-RECORD
         AFTER ADVANCING 1 LINE.
```

7.2 Arithmetic Statements

The five arithmetic statements: ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE, are described in Section 7.6, "PROCEDURE DIVISION Statements," in alphabetical order. For a review of the concepts supported by these statements and the basic rules that govern their use, see Section 2.9, "Arithmetic Statements."

The discussion that follows describes in alphabetical order the optional words that can be used to modify an arithmetic statement.

7.2.1 CORRESPONDING Option

The CORRESPONDING option can reduce the coding required to transfer data during the ADD, SUBTRACT, or MOVE operations.

A pair of data-items from separate identifiers correspond when:

1. the data-items have the same data-name and the same qualifiers up to, but not including their respective identifiers
2. at least one of the data-items is an elementary item (in the case of a MOVE with the CORRESPONDING option)
3. both data-items are elementary numeric data-items (in the case of the ADD and SUBTRACT with CORRESPONDING)

If the proper correspondence exists between the data-items that are specified in an ADD, SUBTRACT, or MOVE statement using the CORRESPONDING option, then the transfer of data from one group item to another will result. The following restrictions apply:

1. At least one of the items specified in a MOVE statement must be an elementary data-item.
2. Both of the items specified in an ADD or SUBTRACT statement must be elementary numeric data-items.
3. The group items must not be described by the USAGE IS INDEX clause or the level-numbers 66, 77, or 88; however, they may contain REDEFINES and OCCURS clauses or be subordinate to data descriptions that do contain these clauses.
4. Corresponding data-items will be ignored if they contain the USAGE IS INDEX, RENAMES, REDEFINES, or OCCURS clauses.

Example:

```
01   ITEM-A.
      05  RECORDS-IN-ERROR     PIC S9(5).
      05  TOTAL-ERRORS         PIC S9(8).
      05  ITEM-A-GROUP.
          10  CORRECTED-RECORDS    PIC S9(8).
          10  RECORD-ID            PIC X(10).
          10  UNIT-PRICE           PIC S9(3)V99.
      05  TOTAL-PRICE          PIC S9(4).

01   ITEM-B.
      05  RECORDS-IN-ERROR     PIC S9(6).
      05  TOTAL-PRICE          PIC S9(4)V99.
      05  R-M-B-NAME           PIC X(10).
      05  ITEM-B-GROUP.
          10 UNIT-PRICE        PIC S9(2)V99.

ADD CORRESPONDING ITEM-A TO ITEM-B.
```

is equivalent to:

```
ADD RECORDS-IN-ERROR OF ITEM-A
    TO RECORDS-IN-ERROR OF ITEM-B.

ADD TOTAL-PRICE OF ITEM-A
    TO TOTAL-PRICE OF ITEM-B.

ADD UNIT-PRICE OF ITEM-A
    TO UNIT-PRICE OF ITEM-B.
```

In this example, data-items RECORDS-IN-ERROR, TOTAL-PRICE, and UNIT-PRICE exist in both ITEM-A and ITEM-B, and will be added. Since TOTAL-RECORDS, CORRECTED-RECORDS, R-M-B-NAME, and RECORD-ID have no corresponding data-item in ITEM-B, they are not involved in the addition. Note that the order in which the CORRESPONDING additions are performed is not defined.

7.2.2 GIVING Option

If the GIVING option is written, the value of the data-name that follows the word GIVING is made equal to the calculated result of the arithmetic operation. The data-name that follows GIVING is not used in the computation and may be a numeric-edited item.

7.2.3 REMAINDER Option

The REMAINDER option directs that a remainder be returned to a specified data field. If the receiving field for the quotient has been defined as numeric-edited, the remainder will be calculated on the quotient's unedited form. If the ROUNDED option has been used on the quotient, the remainder will be calculated on the quotient's truncated form rather than on the rounded form.

7.2.4 ROUNDED Option

If, after decimal-point alignment, the number of places in the fraction of the result is greater than the number of places in the fractional part of the data-item that is to be set equal to the calculated result, truncation occurs unless the ROUNDED option has been specified.

When the ROUNDED option is specified, the least significant digit of the resultant data-name has its value increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

Rounding of a computed negative result is performed by rounding the absolute value of the computed result and then making the final result negative.

The following chart illustrates the relationship between a calculated result and the value stored in an item that is to receive the calculated result, with and without rounding.

Calculated Result	PICTURE	Value After Rounding	Value After Truncation
-12.36	S99V9	-12.4	-12.3
8.432	9V9	8.4	8.4
35.6	99V9	35.6	35.6
65.6	S99V	66	65
.0055	SV999	.006	.005

When the low order integer positions in a receiving field are represented by the character "P" in its PICTURE, rounding or truncation occurs relative to the rightmost integer position for which storage is allowed.

7.2.5 SIZE ERROR Option

If, after decimal-point alignment and any rounding, the value of a calculated result exceeds the largest value that the receiving field is capable of holding, a SIZE ERROR condition exists.

The SIZE ERROR option is written immediately after any arithmetic statement, as an extension of the statement. The format of the SIZE ERROR option is:

```
[ ; ON SIZE ERROR imperative-statement ]
```

If the SIZE ERROR option is present, and a SIZE ERROR condition arises, the value of the resultant data-name is unaltered, and the series of imperative statements specified for the condition is executed.

If the SIZE ERROR option has not been specified and a SIZE ERROR condition arises, no assumption should be made about the final result.

The SIZE ERROR condition applies to the final results of an arithmetic operation. The exceptions are the multiply and divide operations in which the intermediate results are checked for size errors as well as the final results. Within an addition or subtraction operation where the CORRESPONDING phrase has been specified, if any of the individual operations produces a SIZE ERROR condition, the imperative statement in the SIZE ERROR phrase is not executed until all the additions or subtractions are completed.

An arithmetic statement, if written with the SIZE ERROR option, is not an imperative statement. Rather, it is a conditional statement and is prohibited in contexts where only imperative statements are allowed.

7.3 I-O Error Handling

If an I-O error occurs, the file's FILE STATUS data-item, if one exists, is set to the appropriate two-character code. Otherwise, it assumes the value "00".

If an I-O error occurs that is pertinent to the AT END or INVALID KEY condition and an AT END or INVALID KEY phrase exists to handle the error, then the imperative statements in the phrase are executed.

If there is not an appropriate phrase (such phrases may not appear as modifiers to OPEN and CLOSE statements, for example, and are optional for other I-O statements), then the logic of program flow is as follows:

1. If there is an associated DECLARATIVES error procedure, it is performed automatically; user-written logic must determine what action is taken because of the existence of the error. Upon return from the error procedure, normal program flow to the next sentence (following the I-O statement) is allowed.
2. If no DECLARATIVES error procedure is applicable but there is an associated FILE STATUS item, it is presumed that the user may base actions upon testing the status item, so normal flow to the next sentence is allowed. See Chapter 14, "DECLARATIVES Region and USE Statement," for more details about declaring error procedures in the DECLARATIVES Region of a source program.

Only if none of the above (INVALID KEY or AT END phrase, DECLARATIVES error procedure, or testable FILE STATUS data-item) exists, does a runtime error occur.

7.4 Dynamic Debugging Statements

The execution TRACE mode may be set or reset dynamically. When it is set, procedure-names are printed on the user's terminal in the order in which they are executed.

Execution of the READY TRACE statement sets the TRACE mode to cause printing of every section and paragraph name each time it is entered. The RESET TRACE statement inhibits such printing. A printed list of procedure-names in the order of their execution is invaluable in detection of a program malfunction, because it aids in detection of the point at which actual program flow departed from the expected program flow.

Another debugging feature may be required in order to reveal critical data values at specifically designated points in the procedure. The EXHIBIT statement provides this facility. EXHIBIT produces a printout of values of a specified literal or data-item in this format: data – name = value.

The EXHIBIT, READY TRACE, and RESET TRACE statements are extensions to ANSI 74 Standard COBOL. These statements are designed to provide a convenient aid to program debugging. For more information, see the discussions of the individual statements in Section 7.6, "PROCEDURE DIVISION Statements."

Note

It is often desirable to include such statements on source lines that contain D in column 7. In this case, the debugging statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph.

MS-COBOL also provides an interactive debug facility for dynamic program debugging. See the *Microsoft COBOL Compiler User's Guide* for information about this facility.

7.5 MS-COBOL Tape Syntax

While the Microsoft COBOL Compiler recognizes and checks the full language tape-handling syntax, it does not support tape-handling commands during program execution.

7.6 PROCEDURE DIVISION Statements

The remainder of this chapter discusses the individual PROCEDURE DIVISION statements. These statements are arranged alphabetically. For information about how the statements appear in the general format of the PROCEDURE DIVISION, see Section 7.1, "PROCEDURE DIVISION Header and General Format."

7.6.1 ACCEPT Statement

Purpose

The ACCEPT statement is used to obtain low-volume input at runtime, and to place it in a specified receiving field or set of receiving fields.

Format

Four formats are available:

```
ACCEPT identifier FROM { DATE
                       | DAY
                       | TIME
                       | LINE NUMBER
                       | ESCAPE KEY }

ACCEPT identifier [ FROM mnemonic-name ]

ACCEPT (position-spec) identifier [ WITH { ZERO-FILL
                                         | SPACE-FILL
                                         | LEFT-JUSTIFY
                                         | RIGHT-JUSTIFY
                                         | TRAILING-SIGN
                                         | PROMPT
                                         | UPDATE
                                         | LENGTH-CHECK
                                         | AUTO-SKIP
                                         | BEEP
                                         | NO-ECHO
                                         | EMPTY-CHECK } ... ]

ACCEPT screen-name [ ON ESCAPE imperative-statement ]
```

Formats 3 and 4 are Microsoft COBOL extensions to ANSI 74 Standard COBOL. The reserved words LINE NUMBER and ESCAPE KEY in Format 1 are also extensions.

Remarks

The function of each form of the ACCEPT statement is to acquire data from a source external to the program and place it in a specified receiving field or set of receiving fields. The forms differ primarily in the data source with which they are designed to interface.

The Format 1 ACCEPT obtains date or time information from the operating system.

The next two formats of the ACCEPT statement receive data keyed in by an operator at the terminal. For Format 2, this device is assumed to be a teletype, teleprinter, or "dumb" terminal in scrolling mode. For Format 3, it is assumed that the input device is a video terminal capable of direct cursor positioning.

The Format 4 ACCEPT receives an entire data entry form (as defined in the SCREEN SECTION) when it has been completed by the terminal operator. Note that an ordinary terminal is suitable as an input device for a Format 2, 3, or 4 ACCEPT, although the effects on the appearance of the screen will differ. The effects of the various WITH phrase options of the Format 3 ACCEPT statement are summarized in Section 7.6.1.3.

7.6.1.1 Format 1 ACCEPT Statement

Any of several standard values may be obtained at execution time by use of the Format 1 ACCEPT statement.

The formats of the standard values are:

DATE

a six-digit value of the form YYMMDD (year, month, day). Example: July 4, 1976 is 760704.

DAY

a five-digit "Julian date" of the form YYNNN where YY is the two low-order digits of year and NNN is the day-in-year number between 1 and 366.

TIME

an eight-digit value of the form HHMMSSFF where HH is from 00 to 23, MM is from 00 to 59, SS is from 0 to 59, and FF is from 00 to 99; HH is the hour, MM is the minutes, SS is the seconds, and FF represents hundredths of a second.

LINE NUMBER

The ACCEPT...FROM LINE NUMBER statement is provided for compatibility, but in MS-COBOL, the value of LINE NUMBER is always zero.

ESCAPE KEY

a two-digit code generated by the key that terminated the most recently executed Format 3 or Format 4 ACCEPT statement.

Identifier can be interrogated to determine exactly which key was typed. Input may be terminated by any of the following keys, and sets the ESCAPE KEY value as:

Key Name	**Value**
Backtab (terminates only Format 3 ACCEPTs)	99
Escape	01
Field-terminator (of the last field if Format 4 ACCEPT is used)	00
Function key	02-nn

Refer to the *Microsoft COBOL Compiler User's Guide* for information on how key codes are defined for specific terminals. The identifier specified in the format should be an unsigned numeric integer whose USAGE is explicitly or implicitly

DISPLAY, and whose length agrees with the content of the system-defined data-item. If not, the standard rules for a MOVE govern storage of the source value in the receiving item (identifier).

7.6.1.2 Format 2 ACCEPT Statement

Format 2 of the ACCEPT statement is used to accept a string of input characters from a scrolling device such as a teletype or a terminal in scrolling mode. When the ACCEPT statement is executed, input characters are read from the terminal until a carriage return is encountered; then a carriage return/line feed pair is sent back to the console. The input data string is considered to consist of all characters keyed prior to (but not including) the carriage return. If used, the mnemonic-name specified in Format 2 must also be specified in the CONSOLE IS clause of the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION.

For a Format 2 ACCEPT with an alphanumeric receiving field, the input data string is transferred to the receiving field exactly as if it were being MOVEd from an alphanumeric field of length equal to the number of characters in the string. (That is, left justification, space filling, and right truncation occur by default, and right justification and left truncation occur if the receiving field is described as JUSTIFIED RIGHT.) If the receiving field is alphanumeric-edited, it is treated as an alphanumeric field of equal length (as if each character in its PICTURE were "X"), so that no insertion editing will occur.

For a Format 2 ACCEPT with a numeric or numeric-edited receiving field, the input data string is subjected to a validity test which depends on the PICTURE of the receiving field. (If the receiving field is described as COMP-0, its PICTURE is treated as "S9(5)" for purposes of this discussion.) The digits 0 through 9 are considered valid anywhere in the input data string.

The decimal point character is either a period (.) or a comma (,), depending on whether the DECIMAL POINT IS COMMA clause of the CONFIGURATION SECTION is used. In the following discussions, any reference to the decimal point character as a period should be interpreted as if the reference were to a comma if the DECIMAL POINT IS COMMA clause is active.

The decimal point character is considered valid if:

1. it occurs only once in the input data string, and
2. the PICTURE of the receiving field contains a fractional digit position, that is, a 9, Z, *, or floating insertion character which appears to the right of either an assumed decimal point (V) or an actual decimal point (.).

The operational sign characters + and − are considered valid only as the first or last character of the input string and only if the PICTURE of the receiving field contains one of the sign indicators S, +, −, CR, or DB.

All other characters are considered invalid. If the input data string is invalid, the message "INVALID NUMERIC INPUT — PLEASE RETYPE" is sent to the console, and another input data string is read.

When a valid input data string has been obtained, data are transferred to the receiving field exactly as if the instruction being executed were a MOVE to the receiving field from a hypothetical source field with the following characteristics:

1. a PICTURE of the form S9...9V9...9
2. USAGE DISPLAY
3. a total length equal to the number of digits in the input data string
4. as many digit positions to the right of the assumed decimal point as there are digits to the right of the explicit decimal point in the input data string (zero if there is no decimal point in the input data string)
5. current contents equal to the string of digits embedded in the input data string
6. a separate sign with a current negative status if the input data string contains the character "−", and a current positive status otherwise

---

Note

The numeric receiving-field handling described above does not conform to the ANSI 74 COBOL Standard, which treats numeric receiving fields as if they were alphanumeric, justifying data to the left. A runtime switch, described in the *Microsoft COBOL Compiler User's Guide*, is available to make the Format 2 ACCEPT conform to the standard.

7.6.1.3 Format 3 ACCEPT Statement

Format 3 of the ACCEPT statement is used to accept data into a field from a nonscrolling video terminal. The following syntax rules must be observed when the Format 3 ACCEPT is used:

1. The identifier must reference a data-item whose length is less than or equal to 1920 characters.
2. The options SPACE-FILL and ZERO-FILL may not both be specified in the same ACCEPT statement.
3. The options LEFT-JUSTIFY and RIGHT-JUSTIFY may not both be specified within the same ACCEPT statement.
4. If the identifier is described as a numeric-edited item, the UPDATE option must not be specified.
5. The TRAILING-SIGN option may be specified only if the identifier is described as an elementary numeric data-item. If the identifier is described as unsigned, the TRAILING-SIGN option is ignored.
6. For alphanumeric or alphanumeric-edited identifiers, the SPACE-FILL option is assumed if the ZERO-FILL option is not specified, and the LEFT-JUSTIFY option is assumed if the RIGHT-JUSTIFY option is not specified.
7. For numeric or numeric-edited identifiers, the ZERO-FILL option is assumed if the SPACE-FILL option is not specified.

Data Input Field

The position-spec and receiving field (identifier) specifications of the Format 3 ACCEPT statement are used to define the location and characteristics of a data input field on the screen of the terminal.

Location of the Data Input Field

The position-spec is of the form:

```
( [ LIN [ { + } integer-1 ]  ]  , [ COL [ { + } integer-3 ]  ] )
  [       { - }             ]  ,  [       { - }             ]
  [             integer-2   ]     [             integer-4   ]
```

The opening and closing parentheses and the comma separating the two major bracketed groups are required. A space must follow the comma. The position-spec specifies the position on the terminal screen at which the data input field will begin. LIN and COL are MS-COBOL special registers. Each behaves like a numeric data-item with USAGE IS COMP-0, but they may be referred to by every MS-COBOL program without being declared in the DATA DIVISION.

If LIN is specified, the data input field will begin on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by integer-1 if "+ integer-1" (or "− integer-1") is specified. If integer-2 is specified, the data input field will begin on the row whose number is integer-2. If neither LIN nor integer-2 is specified, the data input field will begin on the screen row containing the current cursor position.

If COL is specified, the data input field will begin in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by integer-3 if "+ integer-3" (or "− integer-3") is specified. If integer-4 is specified, the data input field will begin in the screen column whose number is integer-4. If neither COL nor integer-4 is specified, the data input field will begin in the screen column containing the current cursor position.

Characteristics of the Data Input Field

The characteristics (other than position) of the data input field on the terminal screen are determined by the receiving field's PICTURE specification (which is treated as S9(5) in the case of an item whose USAGE is COMP-0). For alphanumeric or alphanumeric-edited identifier-3, the data input field is simply a string of data input character positions starting at the screen location specified by position-spec. The length of the data input field in character positions is equal to the length of the receiving field in memory.

For numeric or numeric-edited identifiers, the data input field may contain any or all of the following: integer digit positions, fractional digit positions, sign position, and decimal point position. There will be one digit position for each 9, Z, *, P, or noninitial floating insertion symbol (a floating insertion symbol is a +, −, or $ which is not the last symbol in a PICTURE character string) in the PICTURE of the identifier.

Each digit position in the data input field is a fractional digit position if the corresponding PICTURE character is to the right of an assumed decimal point (V) or actual decimal point (.) in the PICTURE of the identifier. Otherwise, it is an integer digit position. There will be one sign position if the identifier is described as signed, and no sign position otherwise. There will be one decimal point position if there is at least one fractional digit position, and no decimal point position otherwise.

The data input positions which are defined will occupy successive character positions on the terminal screen, beginning with the position specified by position-spec. If TRAILING-SIGN is specified in the ACCEPT statement, the data input positions will be in the following sequence: integer digit positions (if any), decimal point position (if any), fractional digit positions (if any), and sign position (if any). If TRAILING-SIGN is not specified, the data input positions will be in the following sequence: sign position (if any), integer digit positions (if any), decimal point position (if any), and fractional digit positions (if any).

Data Input and Data Transfer

A character entered into the data input field by the terminal operator may be treated either as an editing character, a terminator character, or a data character. When a terminator key is typed, the ACCEPT is terminated, the data is transferred to the identifier, and the ESCAPE KEY value is set as described in Section 7.6.1.1, "Format 1 Accept Statement." This value can be interrogated by using a Format 1 ACCEPT statement FROM ESCAPE KEY.

The editing characters are DELETE LINE, FORWARD SPACE, BACK SPACE, and DELETE CHARACTER. See the *Microsoft COBOL Compiler User's Guide* to determine which keys perform these functions on your terminal. The action of the editing characters is described later in this section; for now, only data characters will be considered.

Alphanumeric Receiving Field

Consider first the execution of the Format 3 ACCEPT statement with an alphanumeric or alphanumeric-edited receiving field. An alphanumeric-edited receiving field is treated as an alphanumeric field of the same length (as if every character in its PICTURE were "X"). Specifically, no insertion editing will occur.

The initial appearance of the data input field depends on the specifications in the WITH phrase of the ACCEPT statement. If UPDATE is specified, the current contents of the identifier are displayed in the input field. In this case, all data input positions will be treated as if they were keyed by the terminal operator. If UPDATE is not specified, but PROMPT is specified, a period (.) is displayed in each input data position. If neither UPDATE nor PROMPT is specified, the data input field is not changed. The cursor is placed in the first data input position, and characters are accepted as they are keyed by the operator until a terminator character (normally carriage return) is encountered.

If AUTO-SKIP is specified in the ACCEPT statement, the ACCEPT will also be terminated if the operator keys a character into the last (rightmost) data input position.

As each input character is received, it is echoed to the terminal screen. If all positions of the data input field are filled, additional input is ignored until a terminator character or editing character is encountered. If RIGHT-JUSTIFY was specified in the ACCEPT statement, the operator-keyed characters are shifted to the rightmost positions of the data input field when the ACCEPT is terminated. All unkeyed character positions are filled on termination; the fill character is either space (if SPACE-FILL is in effect) or zero (if ZERO-FILL was specified).

The contents of the receiving field will be the same set of characters which appears in the input field; however, the justification of operator-keyed characters will be controlled by the JUSTIFIED specification in the receiving field's data description, not by the RIGHT or LEFT-JUSTIFY option of the ACCEPT statement. Excess positions of the receiving field will be filled with spaces or zeros based on the SPACE-FILL or ZERO-FILL specification in the ACCEPT statement.

Numeric Receiving Field

Next, consider the execution of a Format 3 ACCEPT statement with a numeric or numeric-edited receiving field. As described above, the data input field on the terminal screen may contain integer digit positions, fractional digit positions, or both. First, assume that both are present; the other cases will be treated as variations.

As with the alphanumeric ACCEPT, the data input field may be initialized in a way determined by the WITH options specified in the ACCEPT statement. If UPDATE is specified (not permitted for a numeric-edited receiving field), the integer and fractional parts of the data input field will be set to the integer and fractional parts of the decimal representation of the initial value of the receiving field, with leading and trailing zeros included, if necessary, to fill all digit positions. Except for leading zeros, these initialization characters are treated as operator-keyed data.

When a numeric field with UPDATE is accepted, any digit, sign, or decimal point entered will cause the entire field to be cleared and set to zero or the value of the entered digit. Such a numeric field can be accepted without change by entering a terminator key instead of a digit, sign, or decimal point.

If UPDATE is not specified, but PROMPT is specified, an underscore (_) character will be displayed in each input digit position. In either of these cases (UPDATE or PROMPT), a decimal point will be displayed at the decimal point position.

If neither UPDATE nor PROMPT is specified, the input field on the screen will not be initialized, except for the sign position. The sign position is always initialized positive except when UPDATE is specified, in which case it is initialized according to the sign of the current contents of the receiving field. On most systems, a positive sign position is shown as a space, and a negative sign position is shown as a minus sign.

The cursor is initially placed in the rightmost integer digit position, and characters are accepted one at a time as they are keyed by the operator. A received character may be treated in one of several ways: If the incoming character is a digit, previously keyed digits are shifted one position to the left in the input field and the new digit is displayed in the rightmost integer digit position. If all integer digit positions have not been filled, the cursor remains on the rightmost digit position and another character is accepted. If the entire integer part of the input field has been filled and AUTO-SKIP was specified, the integer part is terminated and the cursor is moved to the leftmost fractional digit position. If the integer part has been filled and AUTO-SKIP was not specified, the cursor is moved to the decimal point position, and any further digits keyed are ignored until the integer part is terminated with a decimal point.

If the character entered is one of the sign characters + or −, the sign position is changed to a positive or negative status, respectively. Cursor position is not affected.

If the character entered is a decimal point character, the integer part is terminated and the cursor is moved to the leftmost fractional digit position.

If the character entered is a field terminator (normally carriage-return), the ACCEPT is terminated and the cursor is turned off. Any other character is ignored.

When the integer part is terminated, the cursor is placed in the leftmost fractional digit position, and operator-keyed characters are again accepted. Digits are simply echoed to the terminal. The sign characters + and – are treated exactly as they were while integer part digits were being entered. The field terminator character terminates the ACCEPT. (If AUTO-SKIP is in effect, filling the entire fractional part also terminates the ACCEPT.) Other characters are ignored. After all digit positions of the fractional part have been filled, further digits are also ignored.

If no fractional digit positions are present, the decimal point is ignored as an input character, and entry of integer part digits may be terminated only by terminating the entire ACCEPT. If no integer digit positions are present, the cursor is initially placed in the leftmost fractional digit position and entry of the fractional part digits proceeds as described above.

On termination of the Format 3 ACCEPT of a numeric or numeric-edited item, data are transferred to the receiving field. The exact form of the data in the receiving field after execution of the ACCEPT is as described in the last paragraph of the discussion of the Format 2 ACCEPT (see Section 7.6.1.2), where the role of the "input data string" mentioned in that paragraph is taken by the string of characters displayed in the data input field. After termination, if SPACE-FILL is in effect, leading zeros in the integer part of the data input field (not in the receiving field) will be replaced by spaces, and the leading operational sign, if present, will be moved to the rightmost space thus created.

Screen Editing Characters With ACCEPT

The editing keys, DELETE LINE, FORWARD SPACE, BACK SPACE, and DELETE CHARACTER, may be used to send screen editing characters. These characters will change data which has already been keyed or supplied by the MS-COBOL runtime system as a result of a WITH UPDATE specification. Entering the DELETE LINE character will cause the ACCEPT to be restarted and all data keyed by the operator or initially present in the receiving field to be lost. The data input field on the terminal screen will be reinitialized if PROMPT is in effect. Otherwise, the data input field will be filled with spaces or zeros according to the SPACE-FILL or ZERO-FILL specification.

Typing the FORWARD SPACE or BACK SPACE characters will move the cursor forward or back one data input position in the case of an alphanumeric or alphanumeric-edited receiving field, or one digit position in the case of a numeric or numeric-edited receiving field. In no case, however, will the FORWARD SPACE or BACK SPACE characters move the cursor outside the range of positions including:

1. the positions already keyed by the operator (or filled by MS-COBOL runtime support when WITH UPDATE is specified), and
2. the rightmost data input position which the cursor has occupied during the execution of this ACCEPT. If the cursor is moved to a position of this range other than the rightmost, and a legal data character is entered, it is displayed at the current cursor position and the cursor is moved forward one data position (alphanumeric or alphanumeric-edited) or one digit position (numeric or numeric-edited).

Typing the DELETE CHARACTER key effectively cancels the last data character entered. The cursor is moved back one data position (digit position if the receiving field is numeric or numeric-edited) and a fill character (space or zero) is displayed under the cursor (except when the cursor is to the left of the decimal point for a numeric ACCEPT). In the case of a numeric field, no fill character is displayed and the cursor is not moved, but the digit at the cursor position is deleted and all digits to the left of it are shifted one position to the right.

Note

The DELETE CHARACTER key has no effect unless the cursor is in position to accept a new data character; in other words, it has no effect if BACK SPACE character(s) have been used to move the cursor back over already keyed positions.

WITH Phrase

The following list summarizes the effects of the WITH phrase specifications for a Format 3 ACCEPT with an alphanumeric or alphanumeric-edited receiving field:

1. SPACE-FILL causes unkeyed character positions of the data input field and the receiving field to be space-filled when the ACCEPT is terminated.
2. ZERO-FILL causes unkeyed character positions of the data input field and the receiving field to be set to ASCII zeros when the ACCEPT is terminated.
3. LEFT-JUSTIFY is treated by this compiler as commentary.
4. RIGHT-JUSTIFY causes operator-keyed characters to occupy the rightmost positions of the data input field after the ACCEPT is terminated. Note that the justification of transferred data in the receiving field is controlled by the JUSTIFIED declaration or default of the receiving field's data description, not by the WITH RIGHT-JUSTIFY phrase.
5. PROMPT causes the data input field on the screen to be set to all periods (.) before input characters are accepted.
6. UPDATE causes the data input field to be initialized with the initial contents of the receiving field and the initial data to be treated as operator-keyed data.
7. LENGTH-CHECK causes a field terminator character to be ignored unless every data input position has been filled.
8. EMPTY-CHECK causes all field terminator characters to be ignored until at least one nonterminator character has been keyed.
9. AUTO-SKIP forces the ACCEPT to be terminated when all data input positions have been filled. A terminator character explicitly keyed has its usual effect.
10. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

11. NO-ECHO causes an asterisk (*) to be displayed for every character entered in the field, rather than displaying the entered character.

The following list summarizes the effects of the WITH phrase specifications for the Format 3 ACCEPT with a numeric or numeric-edited receiving field:

1. SPACE-FILL causes unkeyed digit positions of the data input field (not of the receiving field) to the left of the (possibly implied) decimal point to be space-filled when the ACCEPT is terminated. SPACE-FILL also causes any leading operational sign to be displayed in the rightmost space thus created.
2. ZERO-FILL causes all unkeyed digit positions of the data input field to be set to zero when the ACCEPT is terminated.
3. LEFT-JUSTIFY and RIGHT-JUSTIFY have no effect for a numeric or numeric-edited receiving field.
4. TRAILING-SIGN causes the operational sign to appear as the rightmost position of the data input field. Ordinarily, the sign is the leftmost position of the field.
5. PROMPT causes the data input field positions to be initialized as follows before input characters are accepted: digit positions to underscores (_), decimal point position (if any) to the decimal point character, and sign position (if any) to space.
6. UPDATE causes the data input field to be initialized to the current contents of the receiving field and this initial data to be treated like operator-keyed data.
7. LENGTH-CHECK causes a received decimal point character to be ignored unless all integer digit positions have been keyed, and a field terminator character to be ignored unless all digit positions have been keyed.
8. EMPTY-CHECK causes all field terminator characters to be ignored until at least one nonterminator character has been keyed.

9. AUTO-SKIP causes the integer part of the ACCEPT to be terminated when all integer digit positions have been keyed, and the entire ACCEPT to be terminated when all digit positions have been keyed.

10. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

11. NO-ECHO causes an asterisk (*) to be displayed for every character entered in the field, rather than displaying the entered character.

The following three examples use the Format 3 ACCEPT statement.

Example 1.

Set-up Prior to Executing

Receiving Field: `05 RS-DISCOUNT PIC X(8).`

Initial Contents: `ABCDEFGH`

ACCEPT Statement:
```
ACCEPT (1, 1) RS-DISCOUNT
   WITH PROMPT.
```

Executing the ACCEPT

At Start of ACCEPT: `........`

Operator Enters N: `N.......`

Operator Enters ONE: `NONE....`

Operator Enters Carriage Return: `NONEbbbb`

Result

Final Contents of Receiving Field: `NONEbbbb`

Example 2.

Set-up Prior to Executing

Receiving Field: `10 VEND-NAME PIC X(12).`

Initial Contents: `ACME WIDGETS`

ACCEPT Statement:
```
ACCEPT (1, 1) VEND-NAME
WITH PROMPT UPDATE.
```

At Start of ACCEPT: `ACME WIDGETS`

(If operator enters carriage return here, the receiving field will not be changed.)

Executing the ACCEPT

Operator Enters DELETE LINE: `^..........`

Operator Enters XYZ: `XYZ^.....`

Operator Enters Carriage Return: `XYZbbbbbbbbb`

Result

Final Contents of Receiving Field: `XYZbbbbbbbbb`

Example 3.

Set-up Prior to Executing

Receiving Field: `05 CREDIT PIC S9(4)V99.`

Initial Contents:
```
      +
111111
```

ACCEPT Statement:
```
ACCEPT (LIN + 4, COL - 3) CREDIT
WITH PROMPT TRAILING-SIGN.
```

Executing the ACCEPT

At Start of ACCEPT: `____.__b`

Operator Enters 8: `___8.__b`

Operator Enters 7: `__87.__b`

Operator Enters −: `__87.__-`

Operator Enters 6: `_876.__-`

Operator Enters N: `_876.__-`

Operator Enters . : `_876.__-`

Operator Enters 5: `_876.5_-`

Operator Enters Carriage Return: `_876.5_-`

Result

Final Contents
of Receiving Field: 0876 5$\bar{0}$
(The line above the
low order 0 is used
to indicate the im-
bedded negative sign.)

7.6.1.4 Format 4 ACCEPT Statement

Format 4 of the ACCEPT statement causes a transfer of information from the operator's terminal to all USING and/or TO fields specified in the SCREEN SECTION definition of screen-name or any screen item subordinate to screen-name. Screen items having only VALUE literals or FROM fields or literals have no effect on the operation of the ACCEPT statement. If you wish to have such fields displayed, use the "DISPLAY screen-name" statement before the Format 4 ACCEPT statement.

Each such transfer of data consists of an implicit Format 3 ACCEPT of a field defined by the appropriate screen item's PICTURE followed by an implicit MOVE to the associated TO or USING field.

If the ESCAPE KEY is typed during data input, the entire ACCEPT is terminated without moving the current field to the associated TO or USING item, the ESCAPE KEY value is set to 01, and the imperative-statement in the ON ESCAPE phrase is executed. If a function key is typed, the appropriate ESCAPE KEY value is set and the entire ACCEPT is terminated.

If a field-terminator key (carriage return, tab, etc.,) is typed, the ESCAPE KEY value is set to 00 and the cursor moves to the next input field defined under screen-name, if one exists. If the current field is the last field, the entire ACCEPT is terminated.

If the BACKTAB KEY is typed, the current field is terminated and the cursor moves to the previous input field defined under screen-name. If the current field is the first field, the cursor does not move from that field.

When a field is terminated by a function key, field-terminator key, or BACKTAB KEY, the contents of the current field are moved to the associated TO or USING item, except in the case where no data characters and no editing characters have been entered in that field. This allows the operator to tab forward or backward through the input fields without affecting the contents of the receiving items.

All the editing and validation features described in Section 7.6.1.3 for the Format 3 ACCEPT apply to the Format 4 ACCEPT as well. Several SCREEN SECTION specifications correspond to the Format 3 ACCEPT options:

AUTO	corresponds to	AUTO-SKIP
BELL	corresponds to	BEEP
JUSTIFIED	corresponds to	RIGHT-JUSTIFY
SECURE	corresponds to	NO-ECHO
REQUIRED	corresponds to	EMPTY-CHECK
FULL	corresponds to	LENGTH-CHECK

Furthermore, if an input field specifies the USING clause or both a FROM and TO clause, the ACCEPT will be executed with the UPDATE option. Format 4 ACCEPT statements always use the PROMPT and TRAILING-SIGN options when executing the individual Format 3 ACCEPTs.

If the screen item's PICTURE clause specifies a numeric-edited or alphanumeric-edited input field, the ACCEPT is executed as if the field were numeric or alphanumeric, respectively. When the field is terminated, the data is edited according to the PICTURE clause and redisplayed in the specified screen position. In this case, the JUSTIFIED clause has no effect.

Moves from screen fields to receiving items follow the standard MS-COBOL rules for MOVE statements, except that moves from numeric-edited fields are allowed. In this case, the data is input as if the field were numeric and the MOVE uses only the sign, decimal point, and digit characters.

The Format 4 ACCEPT does not cause the display of any text or prompting label information. That is accomplished by using the "DISPLAY screen-name" statement before accepting the screen-name. See the discussion of DISPLAY in Section 7.6.10 for more information on displaying text.

7.6.2 ADD Statement

Purpose

Adds two or more numeric values and stores the resulting sum.

Format

The general formats for the ADD statement are:

```
ADD   {identifier-1}   [identifier-2]
      {literal-1   }   [literal-2   ]   ... TO identifier-m [ ROUNDED ]

      [ , identifier-n [ ROUNDED ] ] [ ; ON SIZE ERROR imperative-statement ]

ADD   {identifier-1}   {identifier-2}   [identifier-3]
      {literal-1   }   {literal-2   }   [literal-3   ]   ...

      GIVING identifier-m [ ROUNDED ] [ , identifier-n [ ROUNDED ] ]

      [ ; ON SIZE ERROR imperative-statement ]

ADD   {CORRESPONDING}   identifier-1 TO identifier-2 [ ROUNDED ]
      {CORR         }

      [ ; ON SIZE ERROR imperative-statement ]
```

Either the TO or the GIVING option must be specified.

Remarks

When the TO option is used, the values of all the identifiers (including identifier-m, identifier-n...) and literals in the statements are added, and the resulting sum replaces the value of identifier-m, identifier-n, etc. When the GIVING option is used, at least two identifiers and/or numeric literals must be coded between ADD and GIVING. The sum of the values of these identifiers and literals (not including identifier-m, identifier-n...) replaces the value of identifier-m, identifier-n, etc.

When the CORRESPONDING option is used, the corresponding data-items of identifier-1 are added to and stored in the corresponding data-items of identifier-2. Each identifier must refer to a group item, and the composite of operands is determined separately for each pair of corresponding data-items. See Section 7.2.1, "CORRESPONDING Option," for more information.

Neither the composite of operands nor the receiving fields defined in the ADD statement may exceed eighteen (18) decimal digits. This restriction does not apply to the COMPUTE statement. See Section 2.9, "Arithmetic Statements," for a definition of the composite of operands in an arithmetic statement.

Examples

```
ADD INTEREST,
DEPOSIT TO BALANCE ROUNDED.

ADD REGULAR-TIME OVERTIME
GIVING GROSS-PAY.

ADD FEDERAL-TAX, FICA-TAX TO
    DEDUCTIONS, YEAR-TO-DATE-TAX.
```

The last example is equivalent to:

```
ADD FEDERAL-TAX, FICA-TAX TO DEDUCTIONS.
ADD FEDERAL-TAX, FICA-TAX TO YEAR-TO-DATE-TAX.
```

The first statement would result in the sum of INTEREST, DEPOSIT, and BALANCE being placed at BALANCE, while the second would result in the sum of REGULAR-TIME and OVERTIME earnings being placed in item GROSS-PAY. In the third example, the sum of FEDERAL-TAX and FICA-TAX is added to both DEDUCTIONS and YEAR-TO-DATE-TAX.

7.6.3 ALTER Statement

Purpose

Modifies a simple GO TO statement elsewhere in the PROCEDURE DIVISION, thus changing the sequence of execution of program statements.

Format

The general format is:

```
ALTER procedure-name-1 TO [ PROCEED TO ] procedure-name-2

[ , procedure-name-3 TO [ PROCEED TO ] procedure-name-4 ] ...
```

Remarks

Procedure-name-1 and the successive operands in the ALTER statement must refer to MS-COBOL paragraphs. Procedure-name-1 must consist of only a simple GO TO statement; the ALTER statement in effect replaces the former operand of that GO TO by one or more procedure-names.

Note

Since the ALTER statement may easily make a program hard to follow, we *strongly* advise against its use.

Example

```
GATE.
    GO TO MF-OPEN.
MF-OPEN.
    OPEN INPUT MASTER-FILE.
    ALTER GATE TO PROCEED TO NORMAL.
NORMAL.
    READ MASTER-FILE,
    AT END GO TO EOF-MASTER.
```

Examination of the above code reveals the technique of "shutting a gate," providing a one-time initializing program step.

7.6.4 CALL Statement

See Chapter 8, "Interprogram Communication," for a discussion of the CALL and CHAIN statements.

7.6.5 CHAIN Statement

See Chapter 8, "Interprogram Communication," for a discussion of the CALL and CHAIN statements.

7.6.6 CLOSE Statement

The description of the CLOSE statement differs for the various types of file organization. See Chapters 10, 11, and 12 for discussion of CLOSE statements for Sequential, Indexed, and Relative files, respectively.

7.6.7 COMPUTE Statement

Purpose

Evaluates an arithmetic expression and then stores the result in a designated numeric or numeric-edited item.

Format

The general format is:

```
COMPUTE identifier-1 [ ROUNDED ] [ , identifier-2 [ ROUNDED ] ] ...

    = arithmetic-expression [ ; ON SIZE ERROR imperative-statement ]
```

Remarks

Note that exponentiation to an integral power can be accomplished by using the COMPUTE statement.

The COMPUTE statement provides results to arithmetic computations that are not restricted by the "composite of operands" rule. See Section 2.9, "Arithmetic Statements," for a definition of the composite of operands in an arithmetic statement.

Examples

```
COMPUTE GROSS-PAY ROUNDED = BASE-SALARY *
  (1 + 1.5 * (HOURS - 40) / 40).

COMPUTE AMT-CUBED ROUNDED = AMT ** 3.

COMPUTE WS-TOTAL, OUT-TOTAL =
     WS-TOTAL + CURRENT AMT
     ON SIZE ERROR
          PERFORM P800-OVERFLOW.
```

7.6.8 COPY Statement

See Chapter 16 for a discussion of the COPY statement and its use with the REPLACING phrase.

7.6.9 DELETE Statement

See Chapters 11 and 12 for discussion of use of the DELETE statement in Indexed and Relative files.

7.6.10 DISPLAY Statement

Purpose

Provides the capability of outputting low-volume data at run-time without the complexities of file definition.

Format

The general formats are:

```
DISPLAY  { identifier-1 }      [ , identifer-2 ]  ... [ UPON mnemonic-name ]
         { literal-1    }      [ , literal-2   ]

DISPLAY  { (position-spec)  { identifier }  } ...
                            { literal    }
                            { ERASE      }

DISPLAY screen-name
```

See the following remarks for information on individual parts of the format.

Remarks

The following rules must be observed:

1. All identifiers must reference data-items whose lengths are less than or equal to 1920 characters.
2. Mnemonic-name must be defined in the PRINTER IS or CONSOLE IS clause of the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION.
3. Screen-name must be defined in the SCREEN SECTION of the DATA DIVISION.

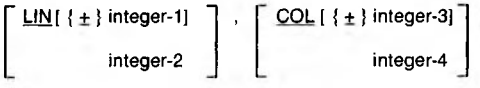
The DISPLAY statement will cause output to be sent to the terminal unless UPON mnemonic-name is specified, in which case output will be sent to the printer. Each display-item (that is, each occurrence of identifier, literal, or ERASE) will be processed in turn as described in the paragraphs below; then, if no

position-spec is coded in the entire DISPLAY statement, a carriage return/line-feed pair will be sent to the receiving device.

Position-Spec

For each display-item, if position-spec is specified, the cursor is positioned prior to the transfer of data for this item.

Position-spec is of the form:



The opening and closing parentheses and the comma separating the two major bracketed groups are required. A space must follow the comma. The position-spec specifies the position on the terminal screen at which the cursor will be placed. LIN and COL are MS-COBOL special registers. Each behaves like a numeric data-item with USAGE IS COMP-0, but they may be referenced by every MS-COBOL program without being declared in the DATA DIVISION. LIN and COL should be initialized before USING. To display the value of LIN or COL, move LIN or COL to a WORKING-STORAGE data-item with PIC 9(5), and display that data-item. LIN and COL do not change as the cursor position changes; they do not represent the current cursor position.

For complete information about cursor positioning by line and column, see Section 7.6.1.3, "Format 3 ACCEPT Statement."

Identifier, Literal, and ERASE

If identifier or literal is specified for a given display-item, the contents of identifier or the value of literal are sent to the receiving device.

If ERASE is specified and if position-spec is coded for this or a previous display-item, the terminal screen will be cleared from the current cursor position to the end of the screen. The initial cursor position for the next display-item will be that specified by the position-spec coded in the ERASE display-item, if

present, or the position in which the cursor was left by the previous display-item. If ERASE is specified and no position-spec has been encountered up to this point in the DISPLAY statement, no action will be taken.

Screen-name

The "DISPLAY screen-name" statement causes a transfer of information from screen-name (or each elementary screen item subordinate to screen-name) to the terminal screen. For each such screen item having a VALUE, FROM, or USING specification, the specified literal or field is the source of the displayed data. For a field having only a TO clause, the effect is as if FROM ALL "." had been specified. The source data is MOVEd implicitly to a temporary item defined by the appropriate screen item's PICTURE (or by the length of the data in the case of a VALUE literal). An implied identifier-type DISPLAY of the constructed temporary is then executed as modified by the positioning and control clause coded in the definition of the appropriate screen item.

Examples

```
DISPLAY QTY-ON-HAND.
DISPLAY INPUT-SCREEN.
DISPLAY (10, 2) USER-NAMES.
DISPLAY (LIN, COL) ERASE.
DISPLAY USER-NAME UPON PRINTER.
```

7.6.11 DIVIDE Statement

Purpose

Divides two numeric values and stores the quotient and the remainder.

Format

The general formats are:

```
DIVIDE   {identifier-1}
         {literal-1   }

    INTO identifier-2 [ ROUNDED ]

        [ , identifier-3 [ ROUNDED ] ] ...

    [ ON SIZE ERROR imperative-statement ]
```

```
DIVIDE   {identifier-1}     {INTO}     {identifier-2}
         {literal-1   }     {BY  }     {literal-2   }

        GIVING   identifier-3 [ ROUNDED ]
                 [ , identifier-4 [ ROUNDED ] ] ...

        [ ON SIZE ERROR imperative-statement ]
```

```
DIVIDE   {identifier-1}     {INTO}     {identifier-2}
         {literal-1   }     {BY  }     {literal-2   }

        GIVING   identifier-3 [ ROUNDED ]

        [ REMAINDER identifier-4 ]

        [ ON SIZE ERROR imperative-statement ]
```

Remarks

The use of BY signifies that the first operand (identifier-1 or literal-1) is the dividend (numerator), and the second operand (identifier-2 or literal-2) is the divisor (denominator). If GIVING is not written in this case, the first operand must be an identifier, in which the quotient is stored. With the GIVING form, multiple results (destinations) are supported.

The use of INTO signifies that the first operand is the divisor and the second operand is the dividend. If GIVING is not written in this case, the second operand must be an identifier, in which the quotient is stored.

The REMAINDER option directs that a remainder be returned to a specified data field. If the receiving field for the quotient has been defined as numeric-edited, the remainder will be calculated on the quotient's unedited form. If the ROUNDED option has been used on the quotient, the remainder will be calculated on the quotient's truncated form rather than the rounded form.

Division by zero always causes a SIZE ERROR condition.

Neither the composite of operands nor the receiving fields defined in the DIVIDE statement may exceed eighteen (18) decimal digits. This restriction does not apply to the COMPUTE statement. See Section 2.9, "Arithmetic Statements," for a definition of the composite of operands in an arithmetic statement.

Examples

```
DIVIDE QTY INTO TOTAL
   GIVING UNIT-COST.

DIVIDE WEIGHT BY 10.

DIVIDE TOTAL-DAYS BY DAYS-IN-WEEK
     GIVING WEEKS-IN-TOTAL
     REMAINDER DAYS-REMAINING
       ON SIZE ERROR
         DISPLAY "DIVISION RESULT TOO LARGE".
```

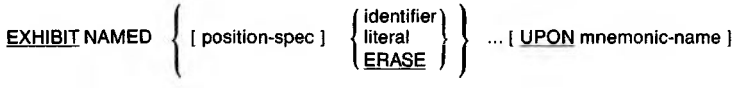
7.6.12 EXHIBIT Statement

Purpose

Displays data values at designated points in a program. This statement is generally used for debugging.

Format

The general format is:



Remarks

EXHIBIT displays the value of the specified literal, or identifiers in this format: identifier = value; i.e., both the value of the identifier and its name are displayed.

The EXHIBIT, READY TRACE, and RESET TRACE statements are extensions to ANSI 74 Standard COBOL. These statements are designed to provide a convenient aid to program debugging.

Position-spec is defined in Section 7.6.10, "DISPLAY Statement."

Note

Including the EXHIBIT, READY TRACE, and RESET TRACE statements on source lines that contain D in column 7 is often desirable. The statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph. See Section 5.2.2, "SOURCE-COMPUTER Paragraph."

For more information on debugging, see Section 7.6.24, "READY/RESET TRACE Statements."

Examples

```
    EXHIBIT QTY-ON-HAND.
D   EXHIBIT DEBUG-VALUES.
```

In the second example, "D" is in column 7, and "EXHIBIT" begins in column 12.

7.6.13 EXIT Statement

Purpose

Provides an end-point for a procedure.

Format

The general format is:

```
EXIT.
```

Remarks

EXIT must appear in the source program as a one-word paragraph preceded by a paragraph-name. An exit paragraph provides an end-point to which preceding statements may transfer control if an operator decides to bypass some part of a section.

Example

```
P100-EXIT-POINT.
    EXIT.
```

7.6.14 EXIT PROGRAM Statement

Purpose

Marks the logical end of a called program.

Format

The general format is:

```
EXIT PROGRAM.
```

Remarks

The EXIT PROGRAM statement must appear in a sentence by itself and must be the only sentence in the paragraph. It is used to terminate a subprogram and to return control to the calling program.

If an EXIT PROGRAM statement is encountered in a program that was not called, the statement is treated as if it were an EXIT statement (see Section 7.6.13, "EXIT Statement").

Example

```
EXIT PROGRAM.
```

7.6.15 GO TO Statement

Purpose

Transfers control from one portion of a program to another.

Format

The general formats are:

```
GO TO [ procedure-name-1 ]

GO TO procedure-name-1 [ , procedure-name-2 ] ..., procedure-name-n

    DEPENDING ON identifier
```

Remarks

The simple form, GO TO procedure-name, changes the path of flow to a designated paragraph or section. If the GO TO statement is without a procedure-name, then that GO TO statement must be the only one in a paragraph, and must be altered prior to its execution.

The more general form designates n procedure-names as a choice of n paths to transfer to, if the value of identifier is 1 to n, respectively. Otherwise, there is no transfer of control, and execution proceeds in the normal sequence. Identifier must be a numeric elementary item and have no positions to the right of the decimal point.

If a GO TO (non-DEPENDING) statement appears in a sequence of imperative statements, it must be the last statement in that sequence.

Examples

```
GO TO P200-PROCESS-RECORD.

GO TO DO-WEEKLY, DO-MONTHLY, DO-YEARLY
    DEPENDING ON MODE-OF-PAYMENT.
```

7.6.16 IF Statement

Purpose

Permits the programmer to specify a series of procedural statements to be executed in the event a stated condition is true. Optionally, an alternative series of statements may be specified for execution if the condition is false.

Format

The general format is:

```
IF condition;  { statement-1      }   { ; ELSE statement-2      }
               { NEXT SENTENCE    }   { ; ELSE NEXT SENTENCE    }
```

Conditional expressions may also be specified in the PERFORM and SEARCH statements. For more details, see Sections 7.6.22 and 9.6, respectively.

Remarks

The ELSE NEXT SENTENCE phrase may be omitted if it immediately precedes the terminal period of the sentence.

Examples

```
IF BALANCE = 0
    PERFORM NOT-FOUND.

IF T LESS THAN 5
   NEXT SENTENCE
ELSE
   PERFORM T-1-4.

IF ACCOUNT-FIELD = SPACES
OR NAME = SPACES
   ADD 1 TO SKIP-COUNT
ELSE
   PERFORM PROCESS-RECORD.
```

```
IF (A + C - 1 = 0)
AND NOT (B NUMERIC)
    PERFORM INPUT-ERROR.
```

The statements following the IF statements are executed only if the designated condition is true. The statements following the ELSE statement are executed only if the designated condition is false.

Regardless of whether the condition is true or false, the next sentence is executed after execution of the appropriate series of statements, unless a GO TO is contained in the imperatives that are executed, or unless the nominal flow of program steps is superseded because of an active PERFORM statement.

If there is no ELSE part to an IF statement, the first series of statements must be terminated by a sentence-ending period. Refer to Appendix B, "Nested IF Statements," for discussion of nested IFs.

7.6.16.1 Methods for Making Comparisons

The discussion that follows describes numeric and character (non-numeric) comparisons.

Numeric Comparisons

The data operands are compared after alignment of their decimal positions. The results are as defined mathematically, with any negative values being less than zero, which in turn is less than any positive value. An index-name or index-data-item may appear in a comparison. Comparison of any two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses, and regardless of length.

Character Comparisons

Non-equal length comparisons are permitted, with spaces being assumed to extend the length of the shorter item, if necessary. Group items are treated simply as characters when compared.

If one operand is numeric and the other is not, the numeric operand must be an integer and have an implicit or explicit USAGE IS DISPLAY. The method of character comparisons is dependent on the current program collating sequence. This will be the standard ASCII sequence or a user-defined alphabet if the alphabet-name phrase and PROGRAM COLLATING SEQUENCE clause were used in the SPECIAL-NAMES and OBJECT-COMPUTER paragraphs, respectively.

If the COLLATING SEQUENCE phrase is used in a SORT or MERGE statement, the collating sequence specified in the phrase becomes the method used for character comparisons during the execution of the SORT or MERGE statement.

Refer to Appendix D for ASCII character representations.

7.6.16.2 Forms of Conditions

Three forms of conditions may be expressed in Microsoft COBOL, including the permissible forms involving parentheses, NOT, or abbreviations. They are:

1. Simple Conditions — The four simple conditions are relational conditions, class conditions, condition-name (level 88) conditions, and sign conditions.
2. Compound Conditions — A compound condition may be formed by connecting two conditions, of any sort, by the logical operator AND or OR (e.g., A = B OR C = D).
3. Complex Conditions — A complex condition exists when simple conditions, compound conditions, and/or other complex conditions are logically connected or negated using the allowable logical operators which are: AND, OR, and NOT.

Simple Conditions

Simple relational condition. Simple relational conditions have three basic forms, expressed by the relational symbols equal to (=), less than (<), or greater than (>).

The six simple relations in conditions are:

Relation	Meaning
=	equal to
<	less than
>	greater than
NOT =	not equal to
NOT <	greater than or equal to
NOT >	less than or equal to

The reserved words EQUAL TO, LESS THAN, and GREATER THAN are accepted equivalents of =, < , and >, respectively. Any form of the relation may be preceded by the optional word, IS.

A simple relational condition has the following structure:

operand-1 relation operand-2

where "operand" is an arithmetic-expression, data-name, literal, or figurative-constant.

Class condition. A class condition has the following format:

```
identifier IS [ NOT ]  { NUMERIC    }
                       { ALPHABETIC }
```

This condition specifies an examination of the data-item to determine whether all characters are proper digit representations regardless of any operational sign (when the test is for NUMERIC), or only alphabetic or blank space characters (when the test is for ALPHABETIC).

The NUMERIC test is valid only for a group, decimal, or character item (not having an alphabetic PICTURE). The ALPHABETIC test is valid only for a group or character item (alphanumeric PICTURE).

Example:

```
IF NUM-VALUE IS NOT ALPHABETIC
    PERFORM NUMERIC-ROUTINE.
```

Sign condition. A sign condition has the following format:

```
arithmetic-expression IS [ NOT ] { POSITIVE }
                                 { NEGATIVE }
                                 { ZERO     }
```

This test is equivalent to comparing an arithmetic expression to zero in order to determine the truth of the stated condition.

Example:

```
IF RECORD-COUNT NOT ZERO
    NEXT SENTENCE
ELSE
    PERFORM INITIALIZE-ROUTINE.
```

Condition-name condition. In a condition-name condition, a conditional variable is tested to determine whether its value is equal to one of the values associated with the condition-name. A condition-name condition is expressed by the following format:

```
condition-name
```

where condition-name is defined by a level 88 DATA DIVISION entry.

Example:

```
    05 END-OF-FILE-SW   PIC X VALUE 'N'.
       88  END-OF-FILE        VALUE 'Y'.
          .
          .
          .
IF END-OF-FILE
    PERFORM EOF-ROUTINE.
```

Compound Conditions

Conditions may be connected to other conditions. The reserved words AND or OR permit the specification of a series of conditions as follows:

1. Individual conditions connected by AND specify a compound condition that is met (true) only if all the individual conditions are true.
2. Individual conditions connected by OR specify a compound condition that is met (true) if any one of the individual conditions is true.

The following examples illustrate a compound condition containing both AND and OR connectors.

```
IF X = Y AND FLAG = 'Z' OR SWITCH = 0
    PERFORM PROCESSING.
```

In the preceding example, execution will be as follows, depending on various data values.

Data X	Value Y	FLAG	SWITCH	Does Execution Go to PROCESSING?
10	10	'Z'	1	Yes
10	11	'Z'	1	No
10	11	'Z'	0	Yes
10	10	'P'	1	No
6	3	'P'	0	Yes
6	6	'P'	1	No

Evaluation rules for compound conditions. The following list presents rules for compound conditions:

1. Evaluation of individual simple conditions (relation, class, condition-name, and sign test) is done first.
2. AND-connected simple conditions are evaluated next as a single result.
3. OR and its adjacent conditions (or previously evaluated results) are then evaluated.

Examples:

1. `A < B OR C = D OR E NOT > F`

   The evaluation is equivalent to (A < B) OR (C = D) OR (E NOT > F) and is true if any of the three individual parenthesized simple conditions is true.

2. `WEEKLY AND HOURS NOT = 0`

   with WEEKLY defined as:

   ```
   05  PAY-CODE     PIC X VALUE SPACE.
       88  WEEKLY          VALUE 'W'.
   ```

   The evaluation is equivalent, after expanding level 88 condition-name WEEKLY, to:

   ```
   (PAY-CODE = 'W') AND (HOURS NOT = 0)
   ```

   and is true only if both the simple conditions are true.

3. ```
   A = 1  AND  B = 2  AND  C = -3
          OR P NOT EQUAL TO "SPAIN"
   ```

   is evaluated as:

   ```
   ((A = 1)  AND  (B = 2)  AND  (C = -3))
   OR  (P NOT = "SPAIN")
   ```

   If P is equal to "SPAIN", the compound condition can only be true if all three of the following are true:

   A = 1
   B = 2
   C = −3

   However, if P is not equal to "SPAIN", the compound condition is true regardless of the values of A, B, and C.

Parenthesized conditions. Parentheses may be written within a compound condition or parts thereof in order to indicate precedence in the evaluation order.

Example:

```
IF  A = B  AND  (C = 5  OR  C = 1)
  PERFORM PROCEDURE-44.
```
```

In this case, PROCEDURE-44 is executed if C = 5 OR C = 1 while at the same time A = B. In this manner, compound conditions may be formed containing other compound conditions, not just simple conditions, with the use of parentheses.

Complex Conditions

The examples that follow illustrate negated and abbreviated conditions.

NOT, the logical negation operator. In addition to its use as a part of a relation (e.g., IF A IS NOT = B), NOT may precede a condition. For example, the condition NOT (A = B OR C) is true when (A = B OR A = C) is false. The word NOT may precede a level 88 condition-name, also.

Note that negating a compound condition may yield unexpected results. For example, the condition

```
NOT (A = B  OR  A = C)
```

is equivalent to:

```
A NOT = B  AND  A NOT = C
```

rather than,

```
A NOT = B  OR  A NOT = C
```

as might be expected.

As an example, assume A = 3, B = 3, and C = 5.

Then, since

A = B is true, and
A = C is false,

1.	A = B OR A = C	is true
2.	NOT (A = B OR A = C)	is false
3.	A NOT = B AND A NOT = C	is false
4.	A NOT = B OR A NOT = C	is true

Conditions 2 and 3 are equivalent. Conditions 2 and 4 are not equivalent.

Abbreviated relational conditions. For the sake of brevity, the user may omit the "subject" when it is common to several successive relational tests. For example, the condition A = 5 OR A = 1 may be written A = 5 OR = 1. This may also be written A = 5 OR 1, where both subject and relation being implied are the same.

Another example:

```
IF  A = B  OR < C OR Y
```

is a shortened form of

```
IF  A = B  OR  A < C OR A < Y
```

The interpretation applied to the use of the word NOT in an abbreviated condition is:

1. If the item immediately following NOT is a relational operator, then the NOT participates as part of the relational operator.
2. Otherwise, the beginning of a new, completely separate condition must follow NOT, not to be considered part of the abbreviated condition.

Warning

Abbreviations in which the subject and relation are implied are permissible only in relation tests; the subject of a sign test or class test cannot be omitted.

7.6.17 INSPECT Statement

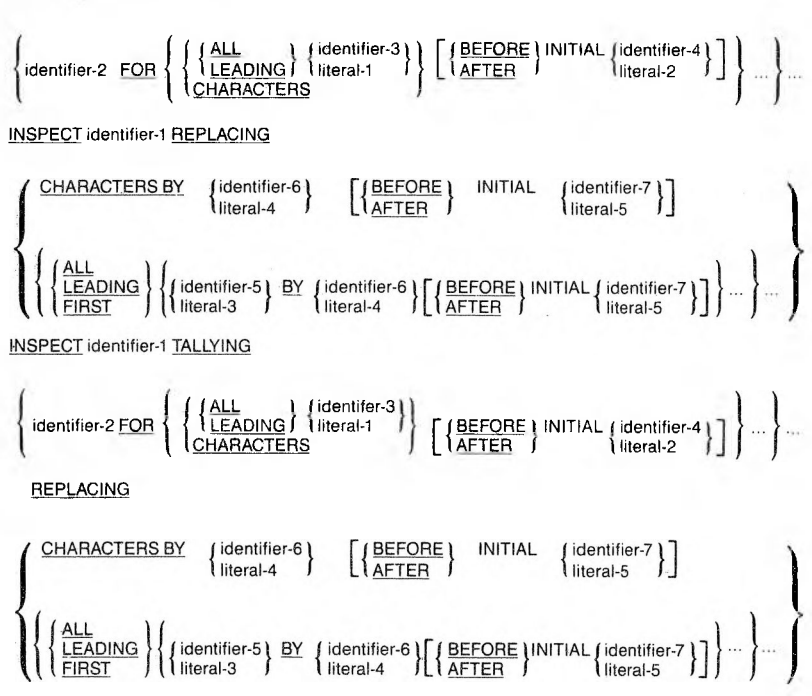
Purpose

Enables the programmer to examine a character-string item. Options permit various combinations of the following actions:

1. Counting appearances of a specified character or character string
2. Replacing a specified character or character string with another
3. Limiting the above actions by requiring the appearance of other specific characters or character strings

Format

The general formats are:



Remarks

In the remarks that follow, operand-n refers to the braced pair which consists of identifier-n and its associated literal; e.g., operand-5 represents {identifier-5 | literal-3}.

Because identifier-1 is to be treated as a string of characters by INSPECT, it must not be described by USAGE IS INDEX, COMP-0, COMP-3, or COMP-4. Identifier-2 must be a numeric data-item.

The TALLYING phrase and REPLACING phrase may not both be omitted; if both are present, the TALLYING phrase must be first.

The TALLYING phrase causes character-by-character comparison, from left to right, of identifier-1, incrementing identifier-2 by one each time a match is found. The matching is done under the following conditions:

1. When an AFTER INITIAL operand-4 subphrase is present, the counting process begins only after detection of a character in identifier-1 matching operand-4.
2. If BEFORE INITIAL operand-4 is specified, the counting process terminates upon encountering a character in identifier-1 which matches operand-4. Also going from left to right, the REPLACING phrase causes replacement of characters under conditions specified by the REPLACING phrase.
3. If BEFORE INITIAL operand-7 is present, replacement does not continue after detection of a character in identifier-1 matching operand-7.
4. If AFTER INITIAL operand-7 is present, replacement does not commence until detection of a character in identifier-1 matching operand-7.

With bounds on identifier-1 thus determined, TALLYING and REPLACING is done on characters as specified by the following:

1. CHARACTERS implies that every character in the bounded identifier-1 is to be TALLYed or REPLACEd.
2. ALL operand-n means that all characters in the bounded identifier-1 which match the operand-n characters are to participate in TALLYING/REPLACING.
3. LEADING operand-n specifies that only characters matching operand-n from the leftmost portion of the bounded identifier-1 which are contiguous (such as leading zeros) are to participate in TALLYING or REPLACING.
4. FIRST operand-n specifies that only the first-encountered characters matching operand-n are to participate in REPLACING. (This option is unavailable in TALLYING.)

When both TALLYING and REPLACING phrases are present, the two phrases behave as if two INSPECT statements were written, the first containing only a TALLYING phrase and the second containing only a REPLACING phrase.

In developing a TALLYING value, the final result in identifier-2 is equal to the tallied count plus the initial value of identifier-2. In the first example below, the item COUNTX is assumed to have been set to zero initially elsewhere in the program.

Examples

```
INSPECT ITEM TALLYING COUNTX
    FOR ALL "L" REPLACING LEADING "A"
    BY "E" AFTER INITIAL "L".
```

Original (ITEM):	SALAMI	ALABAMA
Result (ITEM):	SALEMI	ALEBAMA
Final (COUNTX):	1	1

```
INSPECT WORK-AREA REPLACING ALL DELIMITER
    BY TRANSFORMATION
```

Original (WORK-AREA):	NEW YORK N Y (length 16)
Original (DELIMITER):	(space)
Original (TRANSFORMATION):	. (period)
Result (WORK-AREA):	NEW.YORK..N.Y...

Note

If any identifier-1 or operand-n is described as signed numeric, it is treated as if it were unsigned.

7.6.18 MERGE Statement

The MERGE statement combines two or more identically sequenced files on a set of specified keys. During this process the MERGE statement makes records available in a single merged sequence to an output procedure or to an output file.

See Chapter 13, "SORT/MERGE Facility," for the full description of this statement.

7.6.19 MOVE Statement

Purpose

Moves data from one area of main storage to another and performs conversions and/or editing on the data that is moved.

Format

The general format is:

```
MOVE  { identifier-1 }   TO identifier-2 [ , identifier-3 ] ...
      { literal      }

MOVE  { CORRESPONDING }  identifier-1 TO identifier-2
      { CORR          }
```

In Format 1, the value of the data-item represented by identifier-1 or the specified literal is moved to the receiving fields designated by identifier-2, identifier-3, etc. in the order specified. When a group item is a receiving field, characters are moved without regard to the level structure of the group involved and without editing.

When the CORRESPONDING option is used, identifier-1 and identifier-2 must be group items.

Remarks

Subscripting or indexing associated with receiving fields (identifier-2, identifier-3, etc.) is evaluated immediately before data is moved to the receiving field. Subscripting or indexing associated with a non-literal source field (identifier-1) is evaluated only once, before any data is moved.

The outcome of the data transfer can be adjusted with the JUSTIFIED clause.

To illustrate a data transfer with subscripting, consider the following statement:

```
MOVE A (B) TO B, C (B).
```

This statement is equivalent to:

```
MOVE A (B) TO temp.
MOVE temp TO B.
MOVE temp TO C (B).
```

where temp is an intermediate result field assigned automatically by the compiler. Note that the value of B used as the subscript of A may be different than the value of B used as the subscript of C, since B receives a new value after A (B) is evaluated and before C (B) is evaluated.

Appendix A, "Permissible MOVE Operands," shows, in tabular form, all permissible combinations of source and receiving field types.

The following conditions apply when moving elementary data-items to elementary receiving fields (data-items):

1. If a numeric or alphanumeric data-item is moved to a numeric or numeric-edited data-item:

   a. The source data-item is aligned to the receiving field decimal point, with generation of zeros or truncation on either end, as required. If the source is alphanumeric, it is treated as an unsigned integer. Alphanumeric data-items should not be longer than 31 characters.

   b. When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place. Alphanumeric source items are treated as unsigned integers and are represented in DISPLAY format.

   c. The items may have special editing performed on them such as suppression of zeros and insertion of a dollar sign. Editing characters may replace leading zeros.

   d. Though numeric integers and numeric-edited data-items can be moved to alphanumeric items with or without editing, operational signs are not moved in this case even if SIGN IS SEPARATE has been specified.

2. If an alphabetic or alphanumeric data-item is moved to an alphabetic or alphanumeric-edited item:
   a. The characters are placed in the receiving area from left to right, unless JUSTIFIED RIGHT applies.
   b. If the receiving field is not completely filled by the data being moved, the remaining positions are filled with spaces.
   c. If the source field is longer than the receiving field, the excess characters are truncated on the right after the receiving field is filled.
3. Group item moves are considered alphanumeric; no type conversion is performed.
4. When overlapping fields are involved, results are not predictable.
5. An index-data-item or an index-name cannot appear as an operand of a MOVE statement. See Section 9.4, "SET Statement," for information on modifying index-data-items and index-names.

Examples

The following examples show data movement (a lowercase "b" represents a blank).

Source Field		Receiving Field		
PICTURE	**Value**	**PICTURE**	**Before MOVE**	**After MOVE**
99V99	1234	S99V99	9876-	1234+
99V99	1234	S99V9	987	123
S9V9	12-	99V999	98765	01200
XXX	A2C	XXXXX	Y9X8W	A2Cbb
9V99	123	99.99	87.65	01.23

7.6.20 MULTIPLY Statement

Purpose

Multiplies two numeric data-items and stores the product.

Format

The general formats are:

```
MULTIPLY { identifier-1 } BY identifier-2 [ ROUNDED ]
         { literal-1    }

    [ , identifier-3 [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]

MULTIPLY { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ ROUNDED ]
         { literal-1    }   { literal-2    }

    [ , identifier-4 [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]
```

Remarks

When the GIVING option is omitted, the second operand must be an identifier; the product replaces the value of identifier-2, identifier-3, etc. For example, a new BALANCE value is computed by the statement MULTIPLY 1.03 BY BALANCE. Since this order might seem somewhat unnatural, it is recommended that GIVING always be written.

Neither the composite of operands nor the receiving fields defined in the MULTIPLY statement may exceed eighteen (18) decimal digits. This restriction does not apply to the COMPUTE statement.

See Section 2.9, "Arithmetic Statements," for a definition of the composite of operands in an arithmetic statement.

Examples

```
MULTIPLY UNIT-PRICE BY QTY
    GIVING TOT-PRICE.


MULTIPLY OVERTIME-HOURS BY 1.5
    GIVING CURRENT-OVERTIME, TOTAL-OVERTIME.
```

7.6.21 OPEN Statement

The description of the OPEN statement differs for the various types of file organization. See Chapters 10, 11, and 12 for discussion of OPEN statements for Sequential, Indexed, and Relative files, respectively. For details about the optional file locking syntax for the OPEN statement which is an extension to the full language standard and supports processing in a multi-tasking environment, see Chapter 17, "File and Record LOCKING."

7.6.22 PERFORM Statement

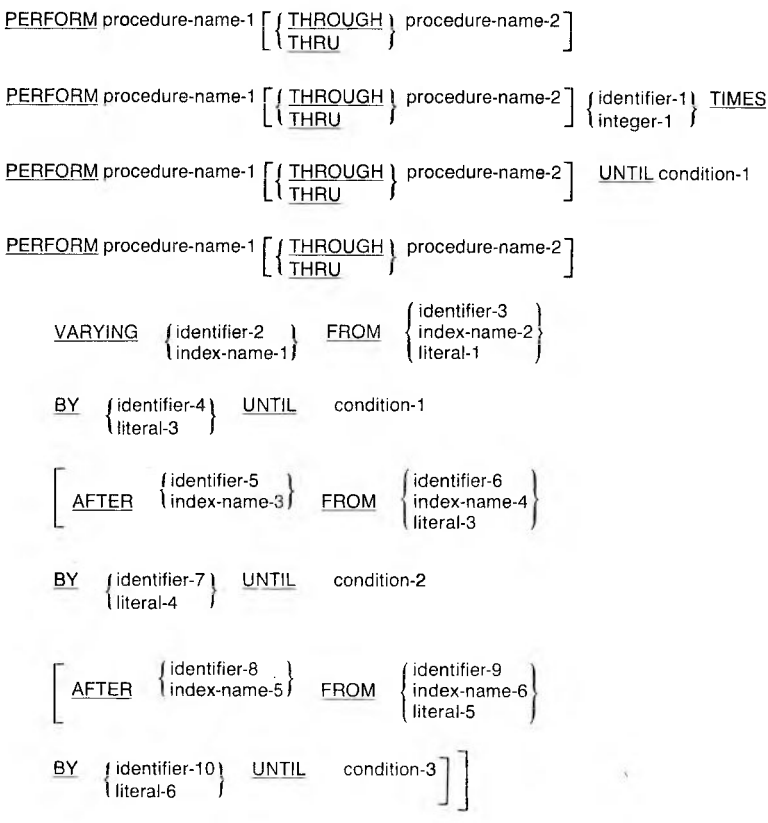
Purpose

Permits the execution of a separate body of program steps.

Format

The range of a PERFORM statement as defined by the "procedure-name" operand, can be a construct of procedure-name-1 THRU procedure-name-2, (THRU being synonymous with THROUGH), or a paragraph-name or a section-name.

The general formats are:



Using Format 2, a designated range may be performed a fixed number of times, as determined by an integer or by the value of an integer data-item within the TIMES phrase. Using Format 3, a designated range may be performed until a conditional value specified in the UNTIL phrase is encountered.

If no TIMES or UNTIL phrase is given, (Format 1 PERFORM), the range is performed once.

Using a Format 4 PERFORM with the VARYING phrase, a range may be performed a variable number of times, with identifier-2 or index-name-1 varying from an initial value of identifier-3 or index-name-2 or literal-1 with increments of literal-4 or literal-3, until a specified condition is met, at which time execution proceeds to the next statement after the PERFORM.

Remarks

If only a paragraph-name is specified as the range of the PERFORM statement, program control returns to the statement following the PERFORM statement after the paragraph's last statement. If only a section-name is specified, control returns after the last statement of the last paragraph of the section. If a construct of procedure-name-1 THRU procedure-name-2 is specified, control is returned after the appropriate last sentence of a paragraph or section.

These return points are valid only when a PERFORM has been executed to set them up; in other cases, control will pass right through. When any PERFORM has finished, execution proceeds to the next statement following the PERFORM.

The condition in a PERFORM using the UNTIL phrase is evaluated prior to each attempted execution of the range. Consequently, it is possible to not PERFORM the range, if the condition is met at the outset. Similarly, if the TIMES phrase is used, and if identifier-1 or integer-1 is less than or equal to 0, the range is not performed at all.

At runtime, it is illegal to have concurrently active PERFORM ranges whose end points are the same.

Examples

```
PERFORM P000-MAINLINE
    THRU P100-WRITE-REPORT.

PERFORM P050-INITIALIZE
    TBL-LENGTH TIMES.

PERFORM P100-WRITE-REPORT
    UNTIL END-OF-FILE.

PERFORM P200-LOOP
   VARYING SUB1 FROM 1 BY 1
   UNTIL (SUB1 > 10
   OR TABLE-VAL (SUB1) = 0).
```

Format 4 PERFORM VARYING can best be described by example. In Example 1a that follows, a PERFORM using a single VARYING clause causes P100-GET-ENTRY to be executed 100 times. Example 1b uses a simple PERFORM and explicit ADD and IF statements, and produces identical results.

Example 1a:

```
P100-MAIN-ENTRY.
     PERFORM P100-GET-ENTRY VARYING ENTRY-COUNT
       FROM 1 BY 1
       UNTIL ENTRY-COUNT GREATER THAN 100.
```

This PERFORM VARYING is equivalent to the following:

Example 1b:

```
     MOVE 1 TO ENTRY-COUNT.

P100-MAIN-ENTRY-LOOP.
     IF ENTRY-COUNT GREATER THAN 100
       GO TO P100-MAIN-ENTRY-EXIT.
     PERFORM P100-GET-ENTRY.
     ADD 1 TO ENTRY-COUNT.
     GO TO P100-MAIN-ENTRY-LOOP.

P100-MAIN-ENTRY-EXIT.
     EXIT.
```

As a more complex example, consider the program fragment in Example 2a. Invoices have been read into working storage, numbered from 100 to 990 in increments of 10. On each invoice, there are 10 lines items which are processed by the routine P400-PROCESS-LINE-ITEM.

Example 2a:

```
P300-PROCESS-DATA.
     PERFORM P400-PROCESS-LINE-ITEM
      VARYING INVOICE-NUMBER FROM 100 BY 10
           UNTIL INVOICE-NUMBER GREATER THAN 990
      AFTER VARYING LINE-ITEM-NUMBER FROM 1 BY 1
           UNTIL LINE-ITEM-NUMBER GREATER THAN 10.
```

This example can be expanded using only PERFORMs with the UNTIL option, and MOVE, ADD, and IF statements:

Example 2b:

```
P300-PROCESS-DATA.
     MOVE 100 TO INVOICE-NUMBER.
     PERFORM P310-INVOICE-LOOP
          UNTIL INVOICE-NUMBER GREATER THAN 990.

P310-INVOICE-LOOP.
     MOVE 1 TO LINE-ITEM-NUMBER.
     PERFORM P320-LINE-ITEM-LOOP
          UNTIL LINE-ITEM-NUMBER GREATER THAN 10.
     ADD 10 TO INVOICE-NUMBER.

P320-LINE-ITEM-LOOP.
     PERFORM P400-PROCESS-LINE-ITEM.
     ADD 1 TO LINE-ITEM-NUMBER.
```

The results of this expanded program fragment and the PERFORM VARYING fragment are identical.

This example can be further expanded using only a simple PERFORM, and MOVE, ADD, IF, and GO TO statements, controlling all looping explicitly:

Example 2c:

```
P300-PROCESS-DATA.
     MOVE 100 TO INVOICE-NUMBER.

P310-INVOICE-LOOP.
     IF INVOICE-NUMBER GREATER THAN 990
          GO TO P340-PROCESS-DATA-EXIT.
     MOVE 1 TO LINE-ITEM-NUMBER.

P320-LINE-ITEM-LOOP.
     IF LINE-ITEM-NUMBER GREATER THAN 10
          GO TO P330-LINE-ITEM-EXIT.
     PERFORM P400-PROCESS-LINE-ITEM.
     ADD 1 TO LINE-ITEM-NUMBER.
     GO TO P320-LINE-ITEM-LOOP.

P330-LINE-ITEM-EXIT.
     ADD 10 TO INVOICE-NUMBER.
     GO TO P310-INVOICE-LOOP.

P340-PROCESS-DATA-EXIT.
     EXIT.
```

Again, the results of the expanded program fragment and the PERFORM VARYING fragment are identical.

7.6.23 READ Statement

The description of the READ statement differs for the various types of file organization. See Chapters 10, 11, and 12 for discussion of READ statements for Sequential, Indexed, and Relative files, respectively.

For details about the optional file locking syntax for the READ statement which is an extension to the full language standard and supports processing in a multi-tasking environment, see Chapter 17, "File and Record LOCKING."

7.6.24 READY/RESET TRACE Statements

Purpose

Execution of a READY TRACE statement sets trace mode to cause printing of every section and paragraph name each time it is encountered. The RESET TRACE statement inhibits such printing.

Format

The general formats for these statements are:

```
RESET TRACE

READY TRACE
```

Remarks

A printed list of procedure-names in the order of their execution can be invaluable in detecting a program error, because it helps find the point at which actual program flow departed from that expected.

The READY TRACE, RESET TRACE, and EXHIBIT statements are extensions to ANSI 74 Standard COBOL. An Interactive Debug Facility with extensive debugging capabilities is also available. See the *Microsoft COBOL Compiler User's Guide* for details.

Note

It is often desirable to include such statements on source lines that contain a "D" in column 7. In this case, the statements are ignored by the compiler unless the WITH DEBUGGING MODE clause is included in the SOURCE-COMPUTER paragraph.

Examples

```
      READY TRACE.
D     RESET TRACE.
```

In the second example, the "D" is in column 7 and "RESET TRACE" begins in column 13.

7.6.25 RELEASE Statement

The RELEASE statement transfers records to the initial phase of a SORT operation.

See Chapter 13, "SORT/MERGE Facility," for the full description of this statement.

7.6.26 RESET TRACE Statement

For a description of the RESET TRACE statement, see Section 7.6.24, "READY/RESET TRACE Statements."

7.6.27 RETURN Statement

The RETURN statement obtains either sorted records from the final phase of a SORT operation or merged records during a MERGE operation.

See Chapter 13, "SORT/MERGE Facility," for the full description of this statement.

7.6.28 REWRITE Statement

The REWRITE statement differs for the various types of file organizations. See Chapter 10, 11, and 12 for discussion of the REWRITE statement in Sequential, Indexed, and Relative files, respectively.

7.6.29 SEARCH Statement

The SEARCH statement is used for the indexing method of table handling. See Chapter 9, "Table Handling by the Indexing Method," for a discussion of this statement.

7.6.30 SET Statement

The SET statement is used for the indexing method of table handling. See Chapter 9, "Table Handling by the Indexing Method," for a discussion of this statement.

7.6.31 SORT Statement

The SORT statement creates a sort file by executing input procedures or by transferring records from one or more USING files.

See Chapter 13, "SORT/MERGE Facility," for the full description of this statement.

7.6.32 START Statement

The START statement is used only with Indexed and Relative files. See Chapters 11 and 12 for discussion of this statement.

For details about the optional file locking syntax for the START statement which is an extension to the full language standard and supports processing in a multi-tasking environment, see Chapter 17, "File and Record LOCKING."

7.6.33 STOP Statement

Purpose

Terminates or delays execution of the object program.

Format

The general format is:

```
STOP  {RUN    }
      {literal}
```

Remarks

STOP RUN terminates execution of a program, returning control to the operating system. If used in a sequence of imperative statements, it must be the last statement in that sequence.

STOP literal displays the specified literal on the terminal and suspends execution.

Execution of the program is resumed only after operator intervention. Presumably, the operator performs a function suggested by the content of the literal prior to resuming program execution by pressing the carriage return key.

Examples

```
CLOSE INV-MSTR-FILE, INV-WARNING-FILE.
STOP RUN.
```

```
STOP "CHANGE DISKETTE,
   THEN PRESS RETURN".
```

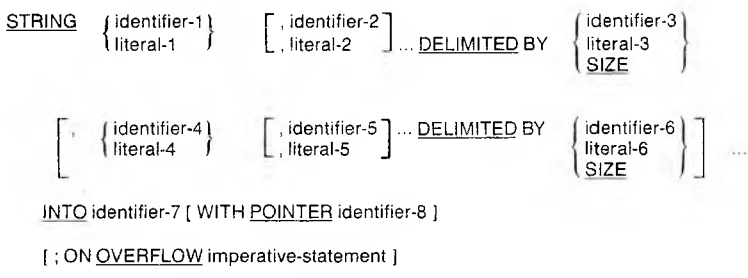
7.6.34 STRING Statement

Purpose

Allows joining together of multiple sending data-item values into a single receiving item.

Format

The general format is:



Remarks

In this format, identifier-7 is the receiving data-item name, which must be alphanumeric without editing symbols or the JUSTIFIED clause; identifier-8 is a counter and must be an elementary numeric integer data-item of sufficient size (plus 1) to point to positions within identifier-7.

If no POINTER phrase exists, the default value of the logical pointer is one. The logical pointer value designates the beginning position of the receiving field into which data placement begins. During movement to the receiving field, the criteria for termination of an individual source are controlled by the DELIMITED BY phrase, as described below.

DELIMITED BY SIZE

The entire source field is moved (unless the receiving field becomes full).

DELIMITED BY an identifier or literal

The character string specified by the identifier or literal is a "key" which, if found to match a like-numbered succession of sending characters, terminates the function for the current sending field (and causes automatic switching to the next sending field, if any).

If at any point the logical pointer (which is automatically incremented by one for each character stored into identifier-7) is less than one or greater than the size of identifier-7, no further data movement occurs, and the imperative statement given in the OVERFLOW phrase (if any) is executed. If there is no OVERFLOW phrase, control is transferred to the next executable statement.

There is no automatic space fill into any position of identifier-7. That is, unaccessed positions are unchanged upon completion of the STRING statement.

Upon completion of the STRING statement, if there was a POINTER phrase, the resultant value of identifier-8 equals its original value plus the number of characters moved during execution of the STRING statement.

Examples

```
STRING OLD-NAME DELIMITED BY SIZE
   INTO NEW-NAME
   WITH POINTER STRING-PTR.

STRING OLD-NAME
   DELIMITED BY STR-DELIM
   INTO NEW-NAME
   ON OVERFLOW
       PERFORM P500-OVERFLOW.

STRING OLD-NAME-1, OLD-NAME-2
   DELIMITED BY SIZE
   INTO NEW-NAME.
```

The following lists show how the values in the last example are affected by the STRING statement.

Variable	PICTURE	SIZE
OLD-NAME-1	PIC X(5)	5
NEW-NAME	PIC X(10)	10
OLD-NAME-2	PIC X(5)	5

For these same variables, the contents are affected as follows:

Variable	Before String	After String
OLD-NAME-1	ABCDE	unchanged
NEW-NAME	1234567	ABCDEFGHIJ
OLD-NAME-2	FGHIJ	unchanged

7.6.35 SUBTRACT Statement

Purpose

Subtracts one or more numeric data-items from a specified item and stores the difference.

Format

The general formats are:

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM identifier-m [ ROUNDED ]
         { literal-1    }  [ , literal-2   ]

  [ identifier-n [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]

SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM { identifier-m }
         { literal-1    }  [ , literal-2   ]         { literal-m    }

GIVING identifier-n [ ROUNDED ]  [ , identifier-o [ ROUNDED ] ... ]

  [ ; ON SIZE ERROR imperative-statement ]

SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2 [ ROUNDED ]
         { CORR          }

  [ ; ON SIZE ERROR imperative-statement ]
```

Remarks

In Format 1, the values of all identifiers and literals preceding FROM are added, and the resulting sum is subtracted from identifier-m, identifier-n, etc. The result of the subtraction replaces the value of identifier-m, identifier-n, etc.

In Format 2, the values of all identifiers and literals preceding FROM are added, and the resulting sum is subtracted from identifier-m or literal-m, and the result is stored as the new value of identifier-n.

When Format 3 is used, data-items of identifier-1 are added to and stored in the corresponding data-items of identifier-2. Each identifier must refer to a group item, and the composite of operands is determined separately for each pair of corresponding data-items. See Section 7.2.1, "CORRESPONDING Option," for more information.

Neither the composite of operands nor the receiving fields defined in the SUBTRACT statement may exceed eighteen (18) decimal digits. This restriction does not apply to the COMPUTE statement.

See Section 2.9, "Arithmetic Statements," for a definition of the composite of operands in an arithmetic statement.

Example

```
SUBTRACT TOT-EXP,
   TOT-DEDUCTIONS FROM TOT-EARNINGS
   GIVING NET-INCOME.
```

7.6.36 UNLOCK Statement

The UNLOCK statement unlocks a record that was previously locked by the execution of a READ or START statement in which the LOCK option was specified. See Chapter 17, "File and Record LOCKING," for details about the use of this statement with Indexed and Relative files.

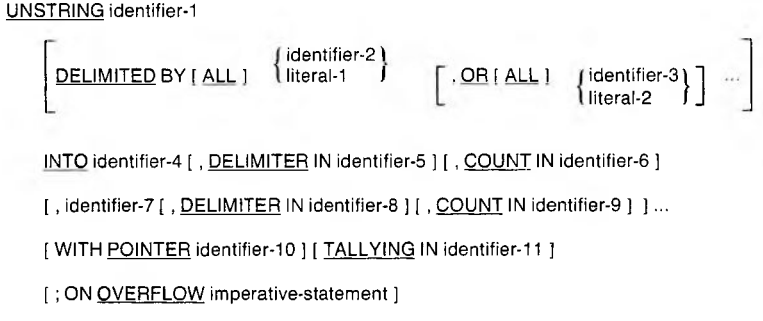
7.6.37 UNSTRING Statement

Purpose

Causes data in a single sending field to be separated into subfields that are placed into multiple receiving fields.

Format

The general format is:



Remarks

The braced items {identifier-2 ! literal-1} and {identifier-3 ! literal-2} may be referred to in the following remarks as operand-i, where "i" refers to the number of the identifier being discussed.

Criteria for separation of subfields may be given in the DELIMITED BY phrase. Each time a succession of characters matches one of the non-numeric literals, one-character figurative constants, or data-item values named by operand-2, the current collection of sending characters is terminated and moved to the next receiving field specified by the INTO clause. When the ALL phrase is specified, more than one contiguous occurrence of operand-2 in identifier-1 is treated as one occurrence.

When two or more delimiters exist, an 'OR' condition exists. Each delimiter is compared to the sending field in the order specified in the UNSTRING statement.

Identifier-1 must be a group or character-string (alphanumeric) item. When a data-item is employed as an operand, that operand must also be a group or character-string item.

Receiving fields (identifiers 4,7,...) may be any of the following types of items:

1. an unedited alphabetic item
2. a character-string (alphanumeric) item
3. a group item
4. an external decimal item (numeric, usage DISPLAY) whose PICTURE does not contain any P character

When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled depending on its type. If there is a DELIMITED BY phrase in the UNSTRING statement, then there may be DELIMITER IN phrases following any receiving item (e.g., identifier-4) mentioned in the INTO clause. In this case, the character(s) that delimit the data moved into identifier-4 are themselves stored in identifier-5, which should be an alphanumeric item. Furthermore, if a COUNT IN phrase is present, the number of characters that were moved into identifier-4 is moved to identifier-6, which must be an elementary numeric integer item.

If there is a POINTER phrase, identifier-10 must be a numeric integer item, and its initial value becomes the initial logical pointer value (otherwise, a logical pointer value of one is assumed). The examination of source characters begins at the position in identifier-1 specified by the logical pointer; upon completion of the UNSTRING statement, the final logical pointer value will be copied back into identifier-10.

If at any time the value of the logical pointer is less than one or exceeds the size of identifier-1, then overflow is said to occur and control passes over to the imperative statements given in the ON OVERFLOW phrase, if any.

Overflow also occurs when all receiving fields have been filled prior to exhausting the source field.

During the course of source field scanning (looking for matching delimiter sequences), a variable length character string is developed which, when completed by recognition of a delimiter or by acquiring as many characters as the size of the current receiving field can hold, is then moved to the current receiving field in the standard MOVE fashion.

If there is a TALLYING IN phrase, identifier-11 must be a numeric integer item. The number of receiving fields acted upon, plus the initial value of identifier-11, will be produced in identifier-11 upon completion of the UNSTRING statement.

Any subscripting or indexing associated with identifier-1, 10, or 11 is evaluated only once at the beginning of the UNSTRING statement. Any subscripting associated with operand-i or identifier-4 through identifier-9 is evaluated immediately before access to the data-item.

Example

```
UNSTRING FIELD-A DELIMITED BY SPACES
   INTO FIELD-B.
```

7.6.38 USE Statement

The USE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the system. The USE statement itself is never executed; it merely defines the procedures that are to be executed under certain conditions.

See Chapter 14, "DECLARATIVES REGION and USE Statement," for details on the USE statement.

7.6.39 WRITE Statement

The WRITE statement differs for the various types of file organizations. See Chapters 10, 11, and 12 for discussion of the WRITE statement in Sequential, Indexed, and Relative files, respectively.

Chapter 8
Interprogram Communication

Separately compiled Microsoft COBOL program modules may be combined into a single executable program. Interprogram communication is made possible through the use of the LINKAGE SECTION of the DATA DIVISION (which follows the WORKING-STORAGE SECTION) and by the CHAIN and CALL statements and the USING and CHAINING phrases of the PROCEDURE DIVISION header.

The LINKAGE SECTION describes data made available in memory from another program module. Record-description entries in the LINKAGE SECTION provide data-names by which data-areas reserved in memory by other programs may be referenced. Entries in the LINKAGE SECTION do not reserve memory areas because the data is assumed to be present in the calling program.

Any record-description entry may be used to describe items in the LINKAGE SECTION as long as the VALUE clause is not specified for other than level 88 items.

8.1 CALL Statement

The CALL statement temporarily transfers control from a program or subprogram to a subprogram.

The format of the CALL statement is:

```
CALL { identifier-1 } [ USING data-name-1 [ , data-name-2 ] ... ]
     { literal-1    }

     [ ; ON OVERFLOW imperative-statement ]
```

Under Microsoft COBOL, COBOL subprograms are not resident in memory until they are called for the first time. When a program is run, only that program (the "main" program) is loaded. When the first CALL to a particular subprogram occurs, COBOL allocates memory for the subprogram and loads it into memory from disk. This is termed "dynamic" loading of subroutines. Subroutines remain in memory until the program terminates, or until explicitly canceled by the program.

When first called, each subprogram is loaded into memory in its initial state. That is, all working storage variables contain data as described in their VALUE clauses, and GO TO statements which might be ALTERed all go to their initial targets.

Subprograms retain their current state on subsequent calls until removed from memory. Working storage data-items may contain data that have been modified because of previous calls. Any GO TO statements retain the target set by the most recent ALTER statement.

Literal-1 is a subprogram name defined as the PROGRAM-ID of a separately compiled program, and must be a non-numeric (quoted) literal. Identifier-1 is a data-item that must be defined as alphanumeric so that its value can be a subprogram name.

See the *Microsoft COBOL Compiler User's Guide* for information and examples for parameter passing and calling of subprograms.

8.1.1 USING Phrase

Data-names in the USING phrase are made available to the called subprogram by passing addresses to the subprogram; these addresses are assigned to the LINKAGE SECTION items declared in the USING phrase of that subprogram. Therefore, the number and order of data-names specified in matching CALL statements and PROCEDURE DIVISION USING phrases must be identical. Information-passing conventions at the machine language level are described in the *Microsoft COBOL Compiler User's Guide.*

Note

Correspondence between caller and callee lists is by position, not by identical spelling of names.

8.1.2 ON OVERFLOW Phrase

If memory is incapable of accommodating the subprogram specified by the CALL statement and the ON OVERFLOW phrase has been used, no transfer of control will occur and the imperative-statement following the ON OVERFLOW phrase will be executed. If the ON OVERFLOW phrase is not present, and memory for the subprogram is not available, the program will terminate abnormally.

8.2 EXIT PROGRAM Statement

The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next executable statement after CALL in the calling program. This statement must be a paragraph by itself. If used in a main program, the EXIT PROGRAM statement is ignored.

8.3 CHAIN Statement

The CHAIN statement, which is an extension to the COBOL standard, causes a specified program to be loaded into memory and executed, replacing the program containing the CHAIN statement.

The CHAIN statement is coded according to the following format:

```
CHAIN { identifier-1 }  [ USING data-name-1 [ , data-name-2 ] ... ]
      { literal-1    }
```

Literal and identifier-1 must be alphanumeric. Each data-name in the USING list must be defined in the WORKING-STORAGE or LINKAGE SECTION or in the record area of a file open at the time the CHAIN statement is executed.

When the CHAIN statement is executed, the value of literal or identifier-1, up to but not including the first space encountered (or the end of the literal or identifier), is interpreted as the name of an executable program in the format of the appropriate operating system or a compiled COBOL program. The named program is loaded into memory and executed.

All program and data structures of the chaining program are permanently destroyed except that the USING phrase may be used to transfer parameters to the chained program. See Section 8.5, "PROCEDURE DIVISION Header With USING/CHAINING Phrases."

The chained program need not be an MS-COBOL program. If it is, it must be a main program.

See the *Microsoft COBOL Compiler User's Guide* for aspects of program chaining and examples of chaining programs which are specific to your operating system.

8.4 CANCEL Statement

The CANCEL statement is used to remove subroutines from memory, freeing that memory for other subprograms, or for other uses. A subsequent CALL to the subprogram causes it to be read into memory from disk again, in its initial state.

The format for the CANCEL statement is:

```
CANCEL  { identifier-1 } [ , identifier-2 ]  ...
        { literal-1    }   , literal-2
```

Identifiers or non-numeric literals specify the subprograms that are to be released.

The identifiers must be defined as alphanumeric data so that the value can be a program name.

Each named and previously-called subprogram is removed from memory. The memory that it occupied is released for use by other functions. If the named subprogram had not been previously called, the respective CANCEL statement is ignored.

A subprogram which is called after having been canceled, will be reloaded into memory in its initial state. Alterable GO TOs and data-items take on their initial values.

Before a valid CANCEL operation can be performed, either the subprogram must never have been called, or an EXIT PROGRAM instruction must have been executed in the subprogram. A subprogram named in the CANCEL statement must not be in the process of being executed.

See Section 8.1, "CALL Statement," for more details.

8.5 PROCEDURE DIVISION Header With USING/CHAINING Phrases

The PROCEDURE DIVISION header of a chained main program or a called subprogram is coded as:

```
PROCEDURE DIVISION  [ { USING    }  data-name-1 [ , data-name-2 ] ... ] ]
                      { CHAINING }
```

where the PROCEDURE DIVISION header of the main program uses CHAINING, and the PROCEDURE DIVISION header of the subprogram uses USING. The forms of the PROCEDURE DIVISION header that use the CHAINING and USING phrases describe the linkage and parameter initialization requirements of a program.

A main program may be run independently or invoked by the execution of a CHAIN statement in another program. A subprogram may only be executed by the action of a CALL statement. See the *Microsoft COBOL Compiler User's Guide* for examples and an explanation of these operations.

Warning

A chained or called program should not have a CHAINING phrase or nonempty USING phrase, unless the invoking CHAIN or CALL statement has a USING list. Furthermore, the numbers of entries in the phrases should be equal. Data entries having corresponding positions in the two lists should refer to data-items of the same size and USAGE.

Failure to conform to these rules will not be detected by the compiler and will cause unpredictable results at runtime.

The values of the data-items named in the PROCEDURE DIVISION header USING or CHAINING phrase are established at program initialization time by using the contents of data-items having corresponding positions in the argument list of the invoking CALL or CHAIN statement. In the case of CALL, the identification is made by passing pointers. Therefore, if the value of a data-item named in a PROCEDURE DIVISION USING phrase is changed during subprogram execution, the corresponding data-item in the calling program will reflect the change after control is returned from the subprogram.

For a description of the formats in which parameters are passed by the CALL and CHAIN statements, see the chapter on interprogram communication in the *Microsoft COBOL Compiler User's Guide.*

Chapter 9
Table Handling by the Indexing Method

Microsoft COBOL supports several methods of referring to elements in tables, using index-names, index-data-items, subscripts, and relative indexes.

9.1 Index-Names and Index-Data-Items

An index-name is declared not by the usual method of level number, name, and data description clauses, but implicitly by appearance in the "INDEXED BY index-name" appendage to an OCCURS clause. An index-name must be unique.

An index-data-item is an item defined by the USAGE IS INDEX clause. An index-data-item must not have a PICTURE clause. An index-name or index-data-item may only be used as an argument in the following contexts:

1. in a SET or SEARCH statement
2. in a CALL statement USING phrase or a PROCEDURE DIVISION header USING phrase
3. in a relational condition
4. as the variation item in a PERFORM VARYING statement
5. in place of a subscript

In all cases, the process is equivalent to dealing with a binary word integer subscript. An index-name may be initialized to some value with a SET, SEARCH, or PERFORM operation.

9.2 Subscripting

Program reference to an item in a table that is controlled by an OCCURS clause is expressed with a proper number of subscripts (or indexes), separated by commas, and enclosed in matching parentheses. For example:

```
TAX-RATE  (BRACKET, DEPENDENTS)
XCODE  (I, 2)
```

Subscripts may be specified as:

1. integer data-names with any USAGE, including USAGE IS INDEX
2. integer numeric literals (for example, 5)
3. index-names

Subscripts may be qualified, but not subscripted. A subscript may be signed, but if so, it must be positive. The lowest acceptable value is 1, pointing to the first element of a table. The highest permissible value is the maximum number of occurrences of the item as specified in its OCCURS clause.

Binary subscripts with USAGE of COMPUTATIONAL-0 or COMPUTATIONAL-4 are recommended for efficiency.

9.3 Relative Indexing

Relative indexing is another method available for referring to elements in a table. In this case, an index is expressed as

index-name (+ or −) integer constant

where a space must be on either side of the plus or minus sign.

For example:

```
XCODE (I + 3, J - 1).
```

9.4 SET Statement

The SET statement permits the manipulation of index-names, index-data-items, or binary subscripts for table-handling purposes.

There are two formats for the SET statement:

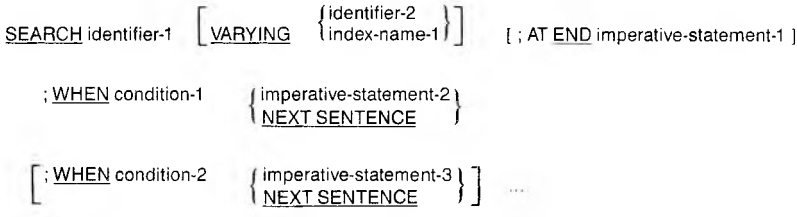
Format 1 is equivalent to moving the TO value (e.g., integer-2) to the multiple receiving fields specified following the verb SET.

Format 2 is equivalent to reduction (DOWN) or increase (UP) applied to each of the quantities specified following the verb SET; the amount of the reduction or increase is specified by a name or value immediately following the word BY. Note that Format 2 is used only with index-names (declared using the INDEXED BY phrase) and not with data-items or identifiers.

In any SET statement, identifiers are restricted to integer data-items.

9.5 Format 1 SEARCH Statement

A linear search of a table may be done using the SEARCH statement. The general format is:



Identifier-1 is the name of a data-item having an OCCURS clause that includes an INDEXED BY phrase; identifier-1 must be written without subscripts or indexes because the nature of the SEARCH statement causes automatic variation of an index-name associated with a particular table.

There are four possible VARYING cases:

1. No VARYING phrase

   The first-listed index-name for the table is varied.

2. VARYING index-name in a different table

   The first-listed index-name in the table's definition is varied, implicitly, and the index-name listed in the VARYING phrase is varied in like manner, simultaneously.

3. VARYING index-name defined for table

   This specific index-name is the only one varied.

4. VARYING integer data-item name

   Both this data-item and the first-listed index-name for the table are varied, simultaneously.

The term "variation" has the following interpretation:

1. The initial value is assumed to have been established by an earlier statement such as SET.

2. If the initial value exceeds the maximum declared in the applicable OCCURS clause, the SEARCH operation terminates at once; if an AT END phrase exists, the associated imperative statement is executed.

3. If the value of the index-name is within the range of valid indexes (1,2, . . . up to and including the maximum number of occurrences), each WHEN condition is evaluated until one is true or all are found to be false. If one is true, its associated imperative statement is executed and the SEARCH operation terminates. If none is true, the index is incremented by one and the steps in this paragraph are repeated. Note that incrementation of the index applies to whatever item and/or index is selected according to the four cases listed above.

If the table is subordinate to another table, an index-name must be associated with each dimension of the entire table via INDEXED BY phrases in all the OCCURS clauses. Only the index-name of the SEARCH table is varied (along with another "VARYING" index-name or data-item). To search an entire two or three-dimensional table, a SEARCH must be executed

several times with the other index-names set appropriately each time, probably with a PERFORM VARYING statement.

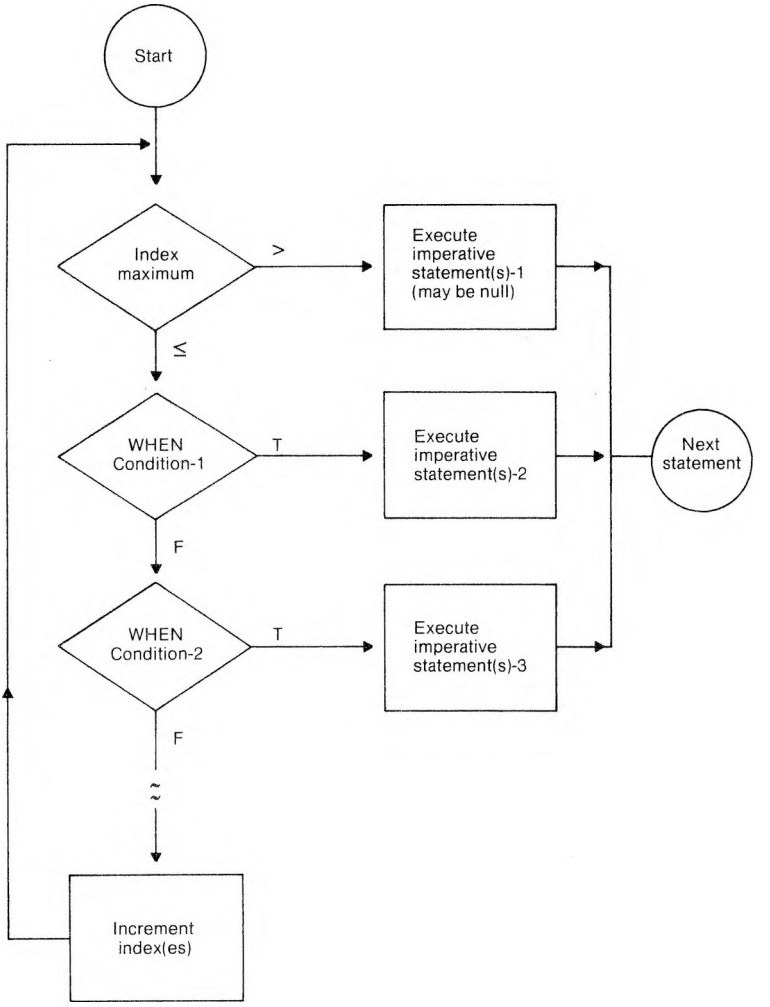
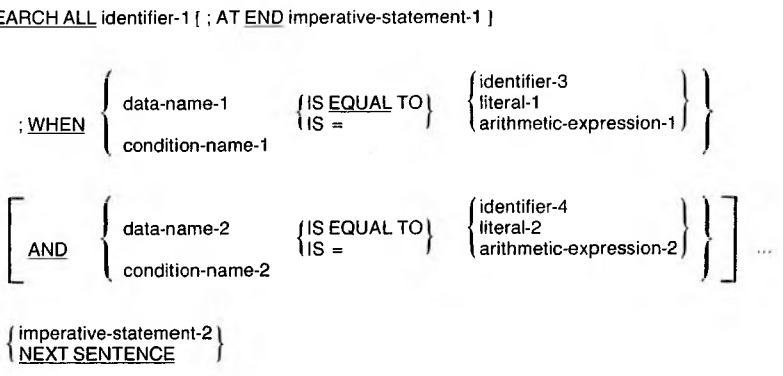


Figure 9.1. Logic Diagram for Format 1 SEARCH Statement

9.6 Format 2 SEARCH Statement

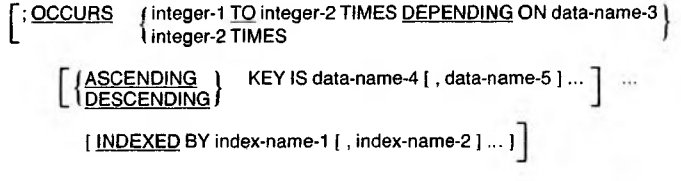
Format 2 SEARCH statements deal with tables of ordered data. The general format of such a SEARCH ALL statement is:



Only one WHEN clause is permitted.

The following rules apply to the condition:

1. Only simple relational conditions or condition-names may be employed, and the subject must be properly indexed by the first index-name associated with identifier-1 (along with sufficient other indexes if multiple OCCURS clauses apply). Furthermore, each subject data-name (or the data-name associated with a condition-name) in the condition must be mentioned in the KEY phrase of the table. The KEY phrase is an appendage to the OCCURS clause having the following format:



where the data-names are the names defined in this data-description entry (following level number) or one of the subordinate data-names. If more than one data-name is given, all of them must be the names of entries subordinate to this group item.

The KEY phrase indicates that the repeated data is arranged in ascending or descending order according to the data-names which are listed (in any given KEY phrase) in decreasing order of significance. More than one KEY phrase may be specified.

2. In a simple relational condition, only the equality test (using relation = or IS EQUAL TO) is permitted.
3. Any condition-name variable (level 88 items) must be defined as having only a single value.
4. The condition may be compounded by use of the logical connector AND, but not OR.
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier. The identifier must not be specified in the KEY phrase of the table or be indexed by the first index-name associated with the table (The term identifier means data-name, including any qualifiers and/or subscripts or indexes.)

Warning

Failure to conform to the restrictions described in the preceding list may yield unpredictable results. Unpredictable results also occur if the table data is not ordered in conformance to the declared KEY phrase, or if the keys referenced in the WHEN-condition are not sufficient to identify a unique table element.

In a Format 2 SEARCH, a nonserial type of search operation may take place, relying upon the declared ordering of data. The initial setting of the index-name for the table is ignored, and its setting is varied automatically during the searching, always within the bounds of the maximum number of occurrences. If the condition (WHEN) cannot be satisfied for any valid index value, control is passed to imperative-statement-1, if the AT END clause is present, or to the next executable sentence in the case of no AT END clause.

If all the simple conditions in the single WHEN condition are satisfied, the resultant index value indicates an occurrence that allows those conditions to be satisfied, and control passes to imperative-statement-2. Otherwise, the final setting is not predictable.

Chapter 10
Sequential Files

This chapter describes SEQUENTIAL file organization and provides source-coding considerations related to its use.

10.1 Definition of SEQUENTIAL File Organization

Records in a file whose ORGANIZATION IS SEQUENTIAL are stored sequentially (in the order in which they were written). These records may be of variable length with each record following the previously written record until the end of the file. This order does not change, except that records may be added to the end of the file.

Sequential files may only be read in the order in which they were written.

Unless otherwise specified, the term "Sequential files" will be used throughout this reference manual, to represent both SEQUENTIAL and LINE SEQUENTIAL file organizations.

There are two organizations of Sequential files:

1. The SEQUENTIAL organization consists of a 2-byte record length followed by the record itself, for as many records as exist in the file. This is the default format for files created by an MS-COBOL program.
2. The LINE SEQUENTIAL organization consists of records followed by delimiters, usually a linefeed or carriage return/linefeed pair, for as many records as exist in the file. See the *Microsoft COBOL Compiler User's Guide* for the delimiters used in your implementation.

   This type of file is often produced by non-COBOL programs, such as text editors. No COMP-0, COMP-3, or COMP-4 data should be written into a Line Sequential file because these data-items may contain the same binary codes used for the record delimiters, and this could subsequently cause problems when the file is read.

Note

If files in Line Sequential format are to be used as input to MS-COBOL programs, the ORGANIZATION IS LINE SEQUENTIAL clause must be specified in the SELECT clause of the input file. If an attempt is made to read a Line Sequential file without this specification, a runtime error will result.

10.2 Syntax Considerations for Sequential File I-O

Information about data file organization and the desired access mode is specified in the ENVIRONMENT DIVISION of a program. Information about the physical characteristics of the data file is specified in the DATA DIVISION. The sections that follow indicate the syntax considerations for the use of SEQUENTIAL and LINE SEQUENTIAL organizations and SEQUENTIAL access mode.

10.2.1 FILE-CONTROL Entry (ENVIRONMENT DIVISION)

The general format for the FILE-CONTROL entry in the ENVIRONMENT DIVISION is:

```
SELECT [ OPTIONAL ] file-name

    ASSIGN TO  { DISK    }
               { PRINTER }

  [ ; [ LOCKING IS ] EXCLUSIVE ]

  [ ; RESERVE integer [ AREA  ] ]
                      [ AREAS ]

    [ ; ORGANIZATION IS [ LINE ] SEQUENTIAL ]

    [ ; ACCESS MODE IS SEQUENTIAL ]

    [ ; FILE STATUS IS data-name-1 ].
```

The SELECT clause must be specified in the FILE-CONTROL paragraph. The clauses which follow the SELECT clause may appear in any order.

The OPTIONAL phrase must be specified if input files are being selected that are not necessarily present each time the object program is executed.

The ORGANIZATION and ACCESS MODE clauses, if not specified, default to ORGANIZATION IS SEQUENTIAL and ACCESS MODE IS SEQUENTIAL, respectively.

For Line Sequential files, the ACCESS MODE clause should be specified as ACCESS MODE IS SEQUENTIAL, or it should be omitted, and the ORGANIZATION IS LINE SEQUENTIAL clause must be in effect.

Note

The record and end-of-file delimiters created by a non-COBOL program must be the same as those used by your implementation of MS-COBOL for proper interpretation of data to occur at runtime. See the *Microsoft COBOL Compiler User's Guide* for the delimiters used with your implementation.

The general formats for the clauses used with Sequential files are given in Chapter 5, "ENVIRONMENT DIVISION."

10.2.2 File Description Entry (DATA DIVISION)

FD (file description) entries in the FILE SECTION are included for each file that was described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. FD entries specify the size of the logical and physical records, the value of implementor-defined label items, names of the data records which make up the file, and the number of lines to be included on a logical printer page. The FD entry is ended with a period (.).

See Section 6.4.1, "FILE SECTION and the File Description (FD) Entry," for more information about file descriptions for Sequential files.

10.2.3 I-O-CONTROL Paragraph (ENVIRONMENT DIVISION)

The I-O-CONTROL paragraph syntax specifies the points at which tape rerun is to be established, the memory area which is to be shared by different files, and the location of files on a multiple file reel.

The general format for the I-O-CONTROL paragraph is:

```
[ I-O-CONTROL.

[ ;RERUN [ ON { file-name-1      } ] EVERY { { { END OF { REEL } } OF file-name-2 } ... } ]
              { implementor-name }         { { { UNIT   } }                       }
                                           { integer-1 RECORDS                    }
                                           { integer-2 CLOCK-UNITS                }
                                           { condition-name                       }

  [ ; SAME [ RECORD ] AREA FOR file-name-3 { , file-name-4 } ... ] ...

  [ ; MULTIPLE FILE TAPE CONTAINS file-name-5 [ POSITION integer-3 ]

        [ , file-name-6 [ POSITION integer-4 ] ] ... ] ... . ]
```

In general, the RERUN clause specifies when and where the rerun information is recorded, the SAME AREA RECORD clause specifies that two or more files are to use the same memory area for processing of the current logical record, and the MULTIPLE FILE clause specifies that more than one file shares the same reel of tape. See Chapter 5, "ENVIRONMENT DIVISION," for details about the MULTIPLE FILE, RERUN, and SAME AREA clauses.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language tape-handling and RERUN syntax, it does not support these commands during program execution.

10.3 File Status Reporting

If the FILE STATUS clause is specified in the FILE-CONTROL paragraph, the designated two-character data-item is set after any CLOSE, OPEN, READ, REWRITE, or WRITE statement and before any USE procedure is executed. The value of this data-item indicates to the program the status of the input-output operation. Table 10.1 summarizes the possible file status settings.

Table 10.1

Sequential File Status Settings

File Status	Meaning
"00"	Successful completion
"10"	End of file
"30"	Permanent error or file not found
"34"	Disk space full
"91"	File structure error
"94"	File locked

In an OPEN INPUT or OPEN I-O statement, a file status of "30" means "File not found."

If file status "91" should occur in a READ statement, it usually indicates that the record size in the file is larger than the size specified in the program record description.

File status "94" occurs when an attempt was made to OPEN a file that is in the locked state because another process had SELECTed it with the LOCKING EXCLUSIVE clause.

10.4 PROCEDURE DIVISION Statements for Sequential Files

The statements that are used to process Sequential and Line Sequential files are:

CLOSE

OPEN

READ

REWRITE

USE

WRITE

The USE statement applies only to Sequential files, and it is used only in the DECLARATIVES Region of the PROCEDURE DIVISION.

The formats and descriptions that apply to Sequential files are given in the remainder of this chapter.

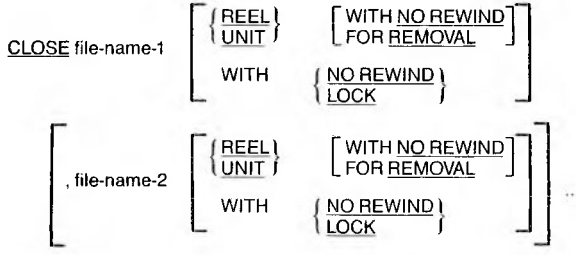
10.4.1 CLOSE Statement

Purpose

Causes the operating system to restore the named file to a storage device. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed; a runtime error would occur, terminating the run.

Format

The general format for SEQUENTIAL and LINE SEQUENTIAL file organizations is:



Remarks

Executed at the end of file processing. If the LOCK suffix is used, the file cannot be reopened during the current job. If LOCK is not specified immediately after a file-name, that file may be reopened later in the program, if program logic dictates the necessity.

Note

The LOCK suffix is a standard COBOL feature, and is not part of the Microsoft COBOL multi-user file locking extensions, which are described in Chapter 17, "File and Record LOCKING."

An attempt to execute a CLOSE statement for a file that is not currently open generates a runtime error, and may cause execution to terminate.

If the OPTIONAL phrase was used in the FILE-CONTROL entry for the file being accessed, but the file did not exist, the standard end-of-file processing for that file is not performed.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language tape-handling syntax, it does not support tape-handling commands during program execution.

Examples

```
CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE.

CLOSE  PRINT-FILE, TAX-RATE-FILE,
       JOB-PARAMETERS WITH LOCK.
```

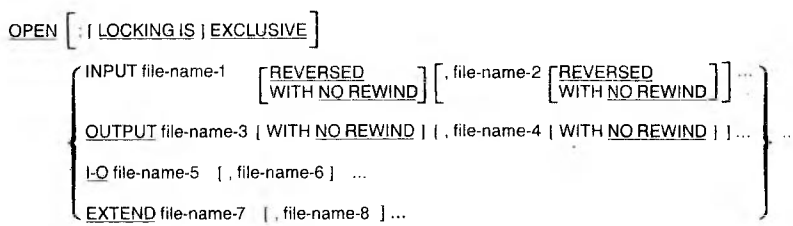
10.4.2 OPEN Statement

Purpose

Prepares a file for use and defines the method of access. The OPEN statement must be executed prior to file processing.

Format

The general format for SEQUENTIAL and LINE SEQUENTIAL file organizations is:



Remarks

OPEN INPUT makes available an area into which an existing file's records may be read. The current record pointer is set to the first record in the file.

OPEN OUTPUT makes available a record area for development of one record, which will be transmitted to the assigned output device upon the execution of a WRITE statement. An existing file which has the same name will be overwritten by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a DISK file; it permits use of the REWRITE statement to modify records which have been accessed by a READ statement. OPEN I-O assumes the existence of the named file, and cannot be used to create the file.

OPEN EXTEND positions the current record pointer at the end of the file for the purpose of appending logical records. Subsequent WRITE statements referencing the file will add records

to the end of the file. Thus, processing proceeds as though the file had been opened with the OUTPUT phrase and positioned at its end.

This mode for initializing a file (OPEN EXTEND) only applies to Sequential and Line Sequential files.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language tape-handling syntax, it does not support tape-handling commands during program execution.

The LOCKING IS EXCLUSIVE clause is optional within the syntax of the SELECT clause and the OPEN statement, and only applies to disk files. EXCLUSIVE is the default of the SELECT clause and the OPEN EXTEND and OPEN OUTPUT modes for SEQUENTIAL and LINE SEQUENTIAL file organizations. For more details, see Chapter 17, "File and Record LOCKING."

The WRITE statement may not be used in I-O mode for files with SEQUENTIAL and LINE SEQUENTIAL organizations.

Failure to precede file reading or writing (in terms of time sequence) by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. Furthermore, a file cannot be opened if it has been closed using the WITH LOCK option.

Sequential files opened for INPUT or I-O access must have been written in the appropriate format described in the *Microsoft COBOL Compiler User's Guide.*

Example

```
OPEN INPUT INV-MSTR-FILE,
   OUTPUT INV-REPORT-FILE.
```

10.4.3 READ Statement

Purpose

Makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data-item, if one was specified.

Format

The general format for SEQUENTIAL and LINE SEQUENTIAL file organizations is:

```
READ file-name RECORD [ INTO identifier ] [ ; AT END imperative-statement ]
```

Since at some time the end-of-file will be encountered, the user should include the AT END phrase.

Remarks

The reserved word END is followed by any number of imperative statements, all of which are executed only if the end-of-file situation arises. The last statement in the AT END series must be followed by a period to indicate the end of the sentence.

If end-of-file occurs but there is no AT END phrase on the READ statement, an applicable DECLARATIVES procedure is performed.

If neither AT END nor DECLARATIVE exists and no FILE STATUS item is specified for the file, a runtime I-O error is processed.

When more than one level 01 item is subordinate to a file definition, these records share the same storage area, and the potential for a record type conflict exists.

The INTO option permits the user to copy a data record into a predefined data field for comparison with the contents in the file's record area. The predefined data field area must not be defined in the FILE SECTION.

Then, using an IF statement to test for a type-field in each record, the user will be able to distinguish between the types of records that are possible, and determine exactly which type is currently available.

The INTO option should not be used when the file has records of various sizes, as indicated by their record descriptions. Any subscripting or indexing of data-name is evaluated after the data has been read but before it is moved to data-name. Afterward, the data is available in both the file record and data-name.

In the case of a blocked input file (such as disk files), not every READ statement performs a physical transmission of data from an external storage device; instead, READ may simply obtain the next logical record from an input buffer.

If the actual record is shorter than the file record area, the file record area is padded on the right with spaces.

When a data record to be read exists, successful execution of the READ statement is immediately followed by execution of the next sentence in the paragraph.

Example

```
READ INV-MSTR-FILE
   INTO WS-MSTR-REC
   AT END MOVE "Y" TO END-OF-FILE-SW.
```

10.4.4 REWRITE Statement

Purpose

Replaces a logical record in a Sequential disk file.

Format

The general format for SEQUENTIAL and LINE SEQUENTIAL file organizations is:

```
REWRITE record-name [ FROM identifier ]
```

Record-name is the name of a logical record in the FILE SECTION of the DATA DIVISION and may be qualified. Record-name and identifier must refer to separate storage areas.

Remarks

At the time of execution of this statement, the file to which record-name belongs must be open in the I-O mode.

If a FROM part is included in this statement, the effect is as if MOVE data-name TO record-name were executed just prior to the REWRITE.

Execution of REWRITE replaces the record that was accessed by the most recent READ statement; the READ must have been completed successfully. If the record which is rewriting the record in the file is longer than the file's record, only as many bytes as will fit are actually rewritten. On the other hand, if the record which is rewriting the record in the file is shorter than the file's record, unpredictable information will be written after the record, until the beginning of the next record in the file.

Example

```
REWRITE PR-REC FROM INV-COUNT.
```

10.4.5 WRITE Statement

Purpose

Releases a logical record for an output or input-output file.

Format

The general format for SEQUENTIAL and LINE SEQUENTIAL file organizations is:

```
WRITE record-name [ FROM identifier-1 ]

[ { BEFORE } ADVANCING { { identifier-2 } [ LINE  ] } ]
  { AFTER  }           { { integer    }   [ LINES ] }
                       { { mnemonic-name }          }
                       { { PAGE          }          }

[ ; AT { END-OF-PAGE } imperative-statement ]
       { EOP         }
```

Remarks

In MS-COBOL, file output is achieved by execution of the WRITE statement. Depending on the device assigned, "written" output may take the form of printed matter or magnetic recording on a floppy disk storage medium. Remember also that you READ file-name, but you WRITE record-name. The associated file must be open in the OUTPUT mode at the time of execution of a WRITE statement.

Record-name must be one of the level 01 records defined for an output file, and may be qualified by the file-name.

If the data to be output has been developed in WORKING-STORAGE or in another area (for example, in an input file's record area), the FROM suffix permits the user to stipulate that the designated data (data-name-1) is to be copied into the record-name area and then output from there. Record-name and data-name-1 must refer to separate storage areas.

When an attempt is made to write beyond the externally defined boundaries of a Sequential file, a DECLARATIVES procedure will be executed (if available) and the FILE STATUS (if available) will indicate a boundary violation. If neither is available, a runtime error occurs.

The ADVANCING phrase is restricted to line printer output files, and permits the programmer to control the line spacing on the paper in the printer. {Identifier-1 | integer} may have values from 0 to 120.

Integer	Carriage Control Action
0	No spacing
1	Normal single spacing
2	Double spacing
3	Triple spacing
.	.
.	.
.	.

Single spacing (i.e., "after advancing 1 line") is assumed if there is no BEFORE or AFTER option in the WRITE statement.

Use of the key word AFTER implies that the carriage control action precedes printing a line, whereas use of BEFORE implies that writing precedes the carriage control action. If PAGE or mnemonic-name are specified, the data is printed BEFORE or AFTER the printer is repositioned to the next physical page. However, if a LINAGE clause is associated with the file, the repositioning is to the first line that can be written on the next logical page as specified in the LINAGE clause.

If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file. EOP is equivalent to END-OF-PAGE.

An end-of-page condition is reached whenever a WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This occurs when such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value specified by the FOOTING value, if specified. In this case, after the WRITE statement is executed, the imperative statement in the END-OF-PAGE phrase is executed.

A page overflow condition is reached whenever a WRITE statement cannot be fully accommodated within the current page body. This occurs when a WRITE statement would cause the LINAGE-COUNTER to exceed the value specified as the size of the page body in the LINAGE clause. In this case, the record is printed before or after (depending on the phrase used) the printer is repositioned to the first line of the next logical page. The imperative statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the printer has been repositioned.

Clearly, if no FOOTING value is specified in the LINAGE clause, or if the end-of-page and overflow conditions occur simultaneously, only the overflow condition is effective.

Example

```
WRITE REPORT-REC FROM PR-HEADER
   AFTER ADVANCING PAGE.
```

Chapter 11
Indexed Files

This chapter describes INDEXED file organization and access, and provides source-coding considerations related to its use.

11.1 Definition of INDEXED File Organization

An Indexed file is organized according to a set of control field values called "keys." These keys are defined in the ENVIRONMENT DIVISION of the source program. An Indexed file must be assigned to DISK in its defining SELECT sentence in the source program's FILE-CONTROL entry.

Each Indexed file declared in a Microsoft COBOL program will generate two disk files: a key file and a data file. The file specification in the VALUE OF FILE-ID clause specifies a file containing data only. The file-name included in the file specification is joined with the extension .KEY to form the file specification of the key file.

The key file contains keys, pointers to keys, and pointers to data. The data file consists of data records and a "data dictionary" containing descriptions of the data records.

A file whose organization is INDEXED can be accessed either sequentially, dynamically, or randomly (see Section 5.3.1.1, "ACCESS MODE Clause").

SEQUENTIAL access provides access to data records in ascending order of RECORD KEY values.

In the RANDOM access mode, the order of access to records is controlled by the programmer. Each record desired is accessed by placing the value of its key in a key data-item prior to an access statement. In the DYNAMIC access mode, the programmer's logic may change from SEQUENTIAL access to RANDOM access, and vice versa, at will.

11.2 Syntax Considerations for Indexed File I-O

Information about data file organization and the desired access mode is specified in the ENVIRONMENT DIVISION of a program. Information about the physical characteristics of the data file is specified in the DATA DIVISION. The sections that follow indicate the syntax considerations for the use of INDEXED organization and SEQUENTIAL, RANDOM, or DYNAMIC access mode.

11.2.1 FILE-CONTROL Entry (ENVIRONMENT DIVISION)

In the FILE-CONTROL entry of the ENVIRONMENT DIVISION, the SELECT sentence must specify ORGANIZATION IS INDEXED, and the file must be assigned to DISK. If an access mode other than SEQUENTIAL is desired during processing, the ACCESS MODE IS clause must be included.

The general format for the FILE-CONTROL entry is:

```
FILE-CONTROL.
SELECT file-name
        ASSIGN TO DISK
       [ ; [ LOCKING IS ]   { EXCLUSIVE }  ]
                            { MANUAL    }
                            { AUTOMATIC }
       [ ; RESERVE integer  [ AREA  ]  ]
                            [ AREAS ]
        ; ORGANIZATION IS INDEXED
       [ ; [ ACCESS MODE IS ]   { SEQUENTIAL }  ]
                                { RANDOM     }
                                { DYNAMIC    }
        ; RECORD KEY IS
                  { data-name-1                                          }
                  { split-key-name-A = data-name-1 [ , data-name-2 ] ... }
      [ ; ALTERNATE RECORD KEY IS
                  { data-name-3       [ WITH DUPLICATES ]  }
                  { split-key-name-B = data-name-3 [ , data-name-4 ] } ]...
                  {                   [ WITH DUPLICATES ]  }
      [ ; FILE STATUS IS data-name-5 ].
```

Dataname-1 through dataname-4 must refer to alphanumeric data-items. Dataname-5 must refer to a two-character alphanumeric data-item defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

11.2.2 RECORD KEY Clause

This clause, which is required, specifies the RECORD KEY that is the prime RECORD KEY for the file. The format for the RECORD KEY clause is as follows:

```
; RECORD KEY IS
      { data-name-1                                          }
      { split-key-name-A = data-name-1 [ , data-name-2 ] ... }
```

The data-item named in the RECORD KEY clause is the prime RECORD KEY for that file. For purposes of inserting, updating, and deleting records in a file, each record is identified solely by the value of its prime RECORD KEY. This value must be unique and must not be changed when updating a file. The key may represent a single field or multiple fields (using the split key syntax). The maximum key length is 250 bytes, and the key value should never be made to contain all binary zeros. A split key is equal to the concatenation of selected data-items.

A record with the following file description entry:

```
FD  EXEMPT-STAFF-FILE
    LABEL RECORD STANDARD
    VALUE OF FILE-ID IS "STAFF.DAT"
    DATA RECORD IS EXEMPT-RECORD.

01  EXEMPT-RECORD.
    02  PRIME-RECORD-KEY      PIC 9(5).
    02  ALT-KEY2-SSNUM        PIC X(20).
    02  ALT-KEY1-NAME         PIC X(20).
    02  PROJECT-ID            PIC X(12).
    02  BONUS-KEY             PIC S9(4)V99.
```

could have this FILE-CONTROL entry:

```
SELECT  EXEMPT-STAFF-FILE
        ASSIGN TO DISK
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS PRIME-RECORD-KEY
        ALTERNATE RECORD KEY IS ALT-KEY2-SSNUM
        ALTERNATE RECORD KEY IS ALT-KEY1-NAME
            WITH DUPLICATES
        ALTERNATE RECORD KEY IS RANK-KEY =
            PRIME-RECORD-KEY
                    PROJECT-ID  BONUS-KEY
        FILE STATUS IS STATUS-INDX.
```

From this control-index structure, the records may be accessed on the prime RECORD KEY, two alternate keys, or a split alternate key which represents three of the fields in the record, including the prime key.

If RANDOM access mode is specified, the value of the primary RECORD KEY designates the record to be accessed by the next DELETE, REWRITE, or WRITE statement.

11.2.3 ALTERNATE RECORD KEY Clause

A data-item named in the ALTERNATE RECORD KEY clause of the FILE-CONTROL paragraph is an alternate RECORD KEY for that file. The key may represent a single field or multiple fields (using the split key syntax).

```
[ ; ALTERNATE RECORD KEY IS

    { data-name-3                [ WITH DUPLICATES ]                 }
    { split-key-name-B =  data-name-3 [ , data-name-4 ]              } ] ...
    {                            [ WITH DUPLICATES ]                 }
```

If the WITH DUPLICATES phrase is used at the end of the ALTERNATE RECORD KEY clause, duplicate field values will be accepted during processing and the alternate RECORD KEY (field value) does not need to be unique.

The DUPLICATES phrase specifies that the value of the associated alternate RECORD KEY may be duplicated within any of the records in the file. If the DUPLICATES phrase is not specified, duplicate values must not occur in the records.

As specified in the FILE-CONTROL paragraph, the key is equal to the concatenation of the selected data-items.

11.2.4 File Description Entry (DATA DIVISION)

FD (file description) entries are included for each file that was described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. FD entries specify the size of the logical and physical records, the value of implementor-defined label items, and the names of the data records which make up the file.

In the FD entry for an Indexed file, both a LABEL RECORDS STANDARD clause and a VALUE OF FILE-ID clause must appear.

11.2.5 I-O-CONTROL Paragraph (ENVIRONMENT DIVISION)

The I-O-CONTROL paragraph syntax specifies the points at which rerun is to be established, and the memory area which is to be shared by different files.

The general format for the I-O-CONTROL paragraph is:

```
[ I-O-CONTROL.

[ ; RERUN ON { file-name-1       } EVERY { integer-1 RECORDS OF file-name-2 } ]
             { implementor-name  }       { integer-2 CLOCK-UNITS           }
                                         { condition-name                  }

[ ; SAME [ RECORD ] AREA FOR file-name-3 { , file-name-4 } ... ] ... . ]
```

In general the RERUN clause specifies when and where the rerun information is recorded, and the SAME AREA RECORD clause specifies that two or more files are to use the same memory area for processing of the current logical record.

See Chapter 5, "ENVIRONMENT DIVISION," for details about the RERUN and SAME AREA clauses.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language RERUN syntax, it does not support RERUN commands during program execution.

11.3 File Status Reporting

If a FILE STATUS clause appears in the ENVIRONMENT DIVISION for an INDEXED organization file, the designated two-character data-item is set after every I-O statement. Table 11.1 summarizes the possible file status settings.

Table 11.1

Indexed File Status Settings

File Status	Meaning
"00"	Successful completion
"02"	Duplicate key; duplicates allowed
"10"	End of file
"21"	Sequence error in writing a Sequential access file
"22"	Duplicate key; duplicates not allowed
"23"	Key not found
"24"	Disk space full
"30"	Permanent error or file not found
"91"	File structure error
"94"	Record or file locked
"95"	Indexed file system not available

File status "02" indicates successful completion of the input-output statement when a duplicate key is used and duplicate keys are allowed (the DUPLICATES phrase is present in the SELECT clause).

1. For a READ statement, the key value for the current key of reference is equal to the value of that same key in the next record within the current key of reference (same field, next record).
2. For a WRITE or REWRITE statement, the record just written creates a duplicate key value in at least one alternate RECORD KEY which allows duplicates.

File status "22" indicates an INVALID KEY condition, and arises if a WRITE or REWRITE statement is executed, the DUPLICATES phrase has been omitted from the SELECT clause, and duplicates are found.

File status "23" arises if the key value specified cannot be found.

In an OPEN INPUT or OPEN I-O statement, file status "30" means "File not found."

File status "91" occurs on an OPEN INPUT or OPEN I-O statement for an Indexed file whose structure has been destroyed (for example, by a system crash during output to the file). When this status is returned the file is not considered to be open, and all I-O operations fail.

File status "94" occurs when an attempt is made to OPEN, READ, or START a READ of a record or file that is in the locked state because another process has SELECTed it with the LOCKING AUTOMATIC or LOCKING MANUAL clause, or has OPENed, READ, or STARTed to READ it with the LOCK option.

File status "95" occurs only in MS-COBOL implementations that separate Indexed file-handler programs (as described in the *Microsoft COBOL Compiler User's Guide*) when the file handler has not been run before Indexed I-O was attempted.

Note

"Disk Space Full" generates file status "24" for Indexed and Relative file handling, whereas it generates file status "34" for Sequential files.

If an error occurs at execution time and no FILE STATUS is specified, no AT END or INVALID KEY phrase imperative statements are given, and no appropriate error-handling sections are supplied in your DECLARATIVES Region, the program will terminate, abnormally.

11.4 PROCEDURE DIVISION Statements for Indexed Files

Table 11.2 summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect. Where X appears, the statement is permissible; otherwise, it is not valid under the associated ACCESS mode and OPEN option. CLOSE is permissible under all conditions.

Table 11.2

I-O Permitted With Indexed Files

		OPEN Option in Effect		
ACCESS MODE IS	**Procedure Statement**	**Input**	**Output**	**I-O**
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

The formats and descriptions that apply to Indexed files are given in the remainder of this chapter.

11.4.1 CLOSE Statement

Purpose

Causes the operating system to restore the named file to a storage device. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed properly; a runtime error will occur, terminating the run.

Format

The general format for INDEXED file organization is:

```
CLOSE file-name-1 [ WITH LOCK ] [ , file-name-2 [ WITH LOCK ] ] ...
```

Remarks

Executed at the end of file processing. If the LOCK suffix is used, the file cannot be reopened during the current job. If LOCK is not specified immediately after file-name, that file may be reopened later in the program, if the program logic dictates the necessity.

Note

The LOCK suffix is a standard COBOL feature, and is not part of the Microsoft COBOL multi-user file locking extensions, which are described in Chapter 17, "File and Record LOCKING."

An attempt to execute a CLOSE statement for a file that is not currently open generates a runtime error, and may cause execution to terminate.

Examples

```
CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE.

CLOSE  PRINT-FILE, TAX-RATE-FILE,
       JOB-PARAMETERS WITH LOCK.
```

11.4.2 DELETE Statement

Purpose

Logically removes a record from the Indexed file.

Format

The general format is:

```
DELETE file-name RECORD [ ; INVALID KEY imperative-statement ]
```

Remarks

For a file in SEQUENTIAL access mode, the last input-output statement executed for file-name would have been a successful READ statement. The record that was read is deleted. Consequently, no INVALID KEY phrase should be specified for files being processed in SEQUENTIAL access mode.

For a file being processed in either RANDOM or DYNAMIC access mode, the record deleted is the one associated with the prime RECORD KEY; if there is no such matching record, an INVALID KEY condition exists, and control passes to the imperative statements in the INVALID KEY phrase, or to an applicable DECLARATIVES Region error-handling section if no INVALID KEY phrase exists.

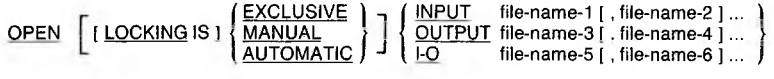
11.4.3 OPEN Statement

Purpose

Prepares a file for use and defines the method of access. The OPEN statement must be executed prior to file processing.

Format

The general format for INDEXED file organization is:



Remarks

A file with INDEXED organization may be opened for access with either SEQUENTIAL, RANDOM, or DYNAMIC access mode, as specified in the ACCESS MODE clause in the program's FILE-CONTROL entry.

OPEN INPUT, with SEQUENTIAL access, sets the current record pointer to the first record of the file. If no records exist for the file, the pointer is set so that the AT END condition will exist on the next (Format 1) READ statement.

OPEN INPUT, with RANDOM or DYNAMIC access, sets the current record pointer to the first record of the file based on the existence of a valid RECORD KEY or other specified valid key (see ALTERNATE RECORD and split key under the FILE-CONTROL entry for Indexed files). If no records exist for the file with the specified key value, an INVALID KEY condition will exist on the next (Format 2) READ statement, and the processing will be altered accordingly.

OPEN I-O assumes the existence of the named file and cannot be used if the file is being created by the program.

OPEN OUTPUT makes available a record area for development of one record, which will be transmitted to the assigned output device upon execution of a WRITE statement. An existing file which has the same name will be overwritten by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a DISK file; it permits use of the REWRITE statement to modify records which have been read into memory by a READ statement.

Failure to precede file reading or writing (in terms of time sequence) by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. A file that has been closed WITH LOCK cannot be opened during the remainder of the program's run.

The LOCKING IS {EXCLUSIVE | MANUAL AUTOMATIC} clause is optional within the syntax of the SELECT clause and the OPEN statement. AUTOMATIC is the default behavior of the SELECT clause and the OPEN I-O and OPEN OUTPUT modes for INDEXED file organization. For more details about locking, see Chapter 17, "File and Record LOCKING."

Example

```
OPEN INPUT INV-MSTR-FILE,
   OUTPUT INV-REPORT-FILE.
```

11.4.4 READ Statement

Purpose

For SEQUENTIAL access, the READ statement makes the next logical record available. For RANDOM access, the READ statement makes a specified record available.

Format

The general format for INDEXED file organization is:

```
READ file-name [ NEXT ] RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
     [ ; AT END imperative-statement ]
READ file-name RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
     [ ; KEY IS data-name ]
     [ ; INVALID KEY imperative-statement ]
```

Format 2 is used for files in RANDOM access mode or for files in DYNAMIC access mode when records are to be retrieved randomly. In Format 2, the INVALID KEY phrase specifies the action to be taken if the key value does not refer to an existing key in the file. If the clause is not given, the appropriate DECLARATIVES Region error-handling section, if supplied, is given control.

If the KEY phrase is used in the Format 2 READ statement, the specified key becomes the key of reference. That key of reference will remain in effect for all subsequent Format 1 READs until another key of reference is specified.

If the KEY phrase is not used in the Format 2 READ statement, the prime RECORD KEY is the key of reference during the DYNAMIC access mode. That key of reference will remain in effect for all subsequent Format 1 READs until another key of reference is specified.

Remarks

Format 1 with the NEXT option is used for sequential reads of a file having DYNAMIC access mode. Format 1 without the NEXT phrase must be used for all files having SEQUENTIAL access mode. The AT END phrase is executed when the logical end-of-file condition arises.

When the AT END condition has been recognized, one of the following statements must be successfully executed before the next Format 1 READ statement execution is attempted:

1. a CLOSE statement followed by the successful execution of an OPEN statement for the file
2. a START statement for the file
3. a Format 2 READ for the same file

In the absence of the AT END phrase, an appropriately assigned DECLARATIVES Region error-handling section is given control at end-of-file time, assuming the section exists.

When an Indexed file is processed sequentially using alternate RECORD KEYs, any records having the same duplicate value are read back in the same order that they were written by the WRITE or REWRITE statements.

The WITH LOCK and WAIT options that have been added to the syntax of the READ and START statements for Indexed and Relative file I-O are optional. For more details about locking, see Chapter 17, "File and Record LOCKING."

Examples

```
READ INV-REC-FILE NEXT RECORD
   INTO REC-COUNT
   AT END PERFORM P300.

READ INV-REC-FILE RECORD INTO REC-COUNT
   KEY IS DATE-REC
   INVALID KEY DISPLAY "REC NOT FOUND".
```

11.4.5 REWRITE Statement

Purpose

Replaces a logical record in the file.

Format

The general format for INDEXED file organization is:

```
REWRITE record-name [ FROM identifier ] [ ; INVALID KEY imperative-statement ]
```

Remarks

For a file being processed in SEQUENTIAL access mode, the last READ statement must have been successful in order for a REWRITE statement to be valid; the record replaced is the one accessed by the READ command.

For a file in RANDOM or DYNAMIC access mode, the record to be replaced is specified by the primary RECORD KEY; no previous READ is necessary. The INVALID KEY condition exists when the value of the key that is the current key of reference does not equal that of any record stored in the file.

The INVALID KEY Condition

In the event that an improper key value is encountered, the system will execute the imperative statements that follow the INVALID KEY phrase; otherwise the error-handling sections of your DECLARATIVES Region are invoked, if applicable. The INVALID KEY condition occurs when:

1. during SEQUENTIAL access, the prime RECORD KEY data-item to be replaced contains a value that is not equal to the prime RECORD KEY of the last record read
2. the value of the prime RECORD KEY data-item is not equal to the primary key field for any record in the file

3. the value of the alternate RECORD KEY data-item equals a corresponding data-item existing in the file; duplicates were not specified

Example

```
REWRITE PR-REC FROM INV-REC
   INVALID KEY PERFORM P300.
```

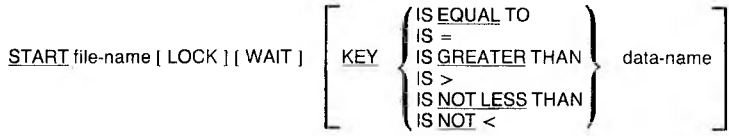
11.4.6 START Statement

Purpose

Enables a file with INDEXED organization to be positioned for reading at a specified key value. This is permitted for files open in either SEQUENTIAL or DYNAMIC access modes.

Format

The general format for INDEXED file organization is:



Data-name must be one of the declared RECORD KEYs.

Remarks

Prior to executing the START statement, the key value that is the object of your search should be placed in the RECORD KEY data-item that will be used for the comparison.

If the key relation is specified as EQUAL TO, the next record accessed will be the first record that is equal to the key value that you specified.

If the key relation is specified as GREATER THAN or NOT LESS THAN, the next record accessed will be the first record that is greater than or equal to the indicated key value.

Establishing the Key of Reference

If the KEY phrase is used, data-name must specify one of the RECORD KEYs in the file. This key is used for the subsequent retrievals. Any subsequent Format 1 READ statements will use this key of reference until another is specified.

If the KEY phrase is not used, the relational operator IS EQUAL TO is implied and the prime RECORD KEY is used for comparison. Any subsequent Format 1 READ statements will use this key of reference until another is specified.

If no matching value is found, the imperative statements in the INVALID KEY phrase are executed. In the absence of this key clause, the imperatives in your DECLARATIVES Region are executed.

The WITH LOCK and WAIT options that have been added to the syntax of the READ and START statements for Indexed and Relative file I-O are optional. For more details about locking, see Chapter 17, "File and Record LOCKING."

Example

```
START INV-REC-FILE
   KEY IS EQUAL TO QTY-RECEIVED
   INVALID KEY DISPLAY "KEY NOT FOUND".
```

11.4.7 UNLOCK Statement

Purpose

Unlocks files formerly locked with a READ or START statement using the WITH LOCK option, or releases the last automatically locked record.

Format

The general format for INDEXED file organization is:

```
UNLOCK file-name
```

Remarks

File-name specifies the file of current reference for either a READ or a START statement. The UNLOCK statement must appear in the same run unit as the READ or a START statement whose operand is the object of its action.

Example

```
UNLOCK MSTR-ACCT-PAYABLE-FILE.
```

11.4.8 WRITE Statement

Purpose

Releases a logical record from the record processing buffer and directs the operating system to transfer the record to the designated output device; this only applies to files in OPEN OUTPUT or OPEN INPUT-OUTPUT mode.

Format

The general format for INDEXED file organization is:

```
WRITE record-name [ FROM identifier ] [ ; INVALID KEY imperative-statement ]
```

Remarks

The value in your prime RECORD KEY must be valid and unique before WRITE statement execution.

The ALTERNATE RECORD KEY value may be non-unique if DUPLICATES were specified for that key.

The INVALID KEY Condition

In the event that an improper key value is encountered, the imperative statements of the INVALID KEY phrase are executed, assuming this clause exists. Otherwise an appropriate DECLARATIVES Region error-handling section is invoked, if applicable.

The INVALID KEY condition exists when

1. for SEQUENTIAL access, key values are not ascending from one WRITE to the next WRITE.
2. the key value is not unique.

3. the allocated disk space is exceeded.
4. the value of the alternate RECORD KEY equals a corresponding data-item existing in the file, and duplicates were not specified.

Example

```
WRITE INV-REC FROM PR-REC
   INVALID KEY DISPLAY "KEY NOT FOUND".
```

Chapter 12
Relative Files

This chapter describes RELATIVE file organization and provides source-coding considerations related to its use.

12.1 Definition of RELATIVE File Organization

RELATIVE file organization is restricted to disk files. Records are differentiated on the basis of a relative record number. Unlike Indexed files, where the identifying key field occupies a part of the data record, relative record numbers are conceptual and are not embedded in the data records. Relative file records are fixed length records whose length is that of the largest record in the file.

A Relative file may be accessed either sequentially, dynamically, or randomly. In SEQUENTIAL access mode, records are accessed in the order of ascending record numbers.

In RANDOM access mode, the sequence of record access is controlled by the program, by placing a number in a relative key item. In DYNAMIC access mode, the program may alternately read a file's records in either RANDOM or SEQUENTIAL access mode.

12.2 Syntax Considerations for Relative File I-O

Information about data file organization and desired access mode is specified in the ENVIRONMENT DIVISION of a program. Information about the physical characteristics of the data file is specified in the DATA DIVISION. The sections that follow indicate the syntax considerations for the use of SEQUENTIAL, RANDOM, or DYNAMIC access mode and RELATIVE organization.

12.2.1 FILE-CONTROL Entry (ENVIRONMENT DIVISION)

In the ENVIRONMENT DIVISION, the FILE-CONTROL entry must specify ORGANIZATION IS RELATIVE. The general format for the SELECT clause is:

```
SELECT file-name
        ASSIGN TO DISK
[ [ LOCKING IS ]    { EXCLUSIVE }  ]
                    { MANUAL    }
                    { AUTOMATIC }
[ ; RESERVE integer  [ AREA  ]  ]
                     [ AREAS ]
  ; ORGANIZATION IS RELATIVE
[ ; ACCESS MODE IS  { SEQUENTIAL [ , RELATIVE KEY IS data-name-1 ] }  ]
                    { { RANDOM  }                                   }
                    { { DYNAMIC } , RELATIVE KEY IS data-name-1     }
  [ ; FILE STATUS IS data-name-2 ].
```

The first byte of the record area associated with a Relative file should not be set to binary zero by being described as part of a COMP-0, COMP-3, or COMP-4 item, nor set to LOW-VALUES by any record description for the file, or the record will be treated as deleted.

In the associated FD entry, both a LABEL RECORDS STANDARD clause and a VALUE OF FILE-ID clause must appear.

12.2.2 RELATIVE KEY Clause

In addition to the usual clauses in the SELECT entry, the "RELATIVE KEY IS data-name-1" clause is required for RANDOM or DYNAMIC access mode. It is also required for SEQUENTIAL access mode, if a START statement exists for such a file.

Data-name-1 must be described as an unsigned integer-item not contained within any record description of the file itself. Its value must be positive and nonzero.

12.2.3 File Description Entry (DATA DIVISION)

FD (file description) entries are included for each file that was described in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION. FD entries specify the size of the logical and physical records, the value of implementor-defined label items, and names of the data records which make up the file.

In the FD entry for a Relative file, both a LABEL RECORDS STANDARD and a VALUE OF FILE-ID clause must appear.

12.2.4 I-O-CONTROL Paragraph (ENVIRONMENT DIVISION)

The I-O-CONTROL paragraph syntax specifies the points at which RERUN is to be established, and the memory area which is to be shared by different files.

The general format for the I-O-CONTROL paragraph is:

```
[ I-O-CONTROL.

    [ ; RERUN ON { file-name-1       } EVERY { integer-1 RECORDS OF file-name-2 } ] ...
                 { implementor-name  }       { integer-2 CLOCK-UNITS           }
                                             { condition-name                  }

    [ ; SAME [ RECORD ] AREA FOR file-name-3 [ , file-name-4 ] ... ] ... ]
```

In general, the RERUN clause specifies when and where the rerun information is recorded, and the SAME AREA RECORD clause specifies that two or more files are to use the same memory area for processing of the current logical record.

See Chapter 5, "ENVIRONMENT DIVISION," for details about the RERUN and SAME AREA clauses.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language RERUN syntax, it does not support RERUN commands during program execution.

12.3 File Status Reporting

If a FILE STATUS clause appears in the ENVIRONMENT DIVISION for a Relative file, the designated two-character data-item is set after every I-O statement. Table 12.1 summarizes the possible file status settings:

Table 12.1

Relative File Status Settings

File Status	Meaning
"00"	Successful completion
"10"	End of file
"22"	Duplicate key
"23"	Key not found
"24"	Disk space full
"30"	Permanent error
"94"	Record or file locked

In an OPEN INPUT or OPEN I-O statement, a file status of "30" means "File Not Found."

File status "94" occurs when an attempt is made to OPEN, READ, or START a READ of a record or file that is in the locked state because another process has SELECTed it with the LOCKING AUTOMATIC or LOCKING MANUAL clause, or has OPENed, READ, or STARTed to READ it with the LOCK option.

"Disk Space Full" generates file status "24" for Relative and Indexed file handling, whereas it generates file status "34" for Sequential files.

12.4 PROCEDURE DIVISION Statements for Relative Files

Within the PROCEDURE DIVISION, the verbs OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, and START are available, just as for files whose organization is INDEXED.

The formats for OPEN and CLOSE are the same as those described in Chapter 7, "PROCEDURE DIVISION," except that the EXTEND phrase is NOT applicable to the OPEN statement for Relative files.

Table 12.2 summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect. An X indicates that the statement is permissible. CLOSE is permissible under all conditions.

Table 12.2

I-O Permitted With Relative Files

ACCESS MODE IS	Procedure Statement	OPEN Option in Effect: Input	Output	I-O
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

The formats and descriptions that apply to Relative files are given in the remainder of this chapter.

12.4.1 CLOSE Statement

Purpose

Causes the operating system to restore the named file to a storage device. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed properly; a runtime error will occur, terminating the run.

Format

The general format for RELATIVE file organization is:

```
CLOSE file-name-1 [ WITH LOCK ] [ , file-name-2 [ WITH LOCK ] ] ...
```

Remarks

Executed at the end of file processing. If the LOCK suffix is used, the file cannot be reopened during the current job. If LOCK is not specified immediately after a file-name, that file may be reopened later in the program, if program logic dictates the necessity.

Note

The LOCK suffix is a standard COBOL feature, and is not part of the Microsoft COBOL multi-user file locking extensions, which are described in Chapter 17, "File and Record LOCKING."

An attempt to execute a CLOSE statement for a file that is not currently open generates a runtime error, and may cause execution to stop.

Examples

```
CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE.

CLOSE  PRINT-FILE, TAX-RATE-FILE,
       JOB-PARAMETERS WITH LOCK.
```

12.4.2 DELETE Statement

Purpose

Removes a record from the file. The record that is deleted is the last one that was read.

Format

The format of the DELETE statement is the same for a Relative file as it is for an Indexed file:

```
DELETE file-name RECORD [ ; INVALID KEY imperative-statement ]
```

Remarks

For a file in SEQUENTIAL access mode, the last input-output statement executed for file-name would have been a successful READ statement. The record that was read is deleted. Consequently, no INVALID KEY phrase should be specified for files being processed in SEQUENTIAL access mode.

For a file being processed in either RANDOM or DYNAMIC access mode, the record deleted is the one associated with the RELATIVE KEY; if there is no such matching record, an INVALID KEY condition exists, and control passes to the imperative statements in the INVALID KEY phrase, or to an applicable DECLARATIVES Region error-handling section if no INVALID KEY phrase exists.

Example

```
DELETE INV-REC RECORD
   INVALID KEY DISPLAY "KEY NOT FOUND".
```

12.4.3 OPEN Statement

Purpose

Prepares a file for use and defines the method of access. The OPEN statement must be executed prior to file processing.

Format

The general format for RELATIVE file organization is:

Remarks

A file with RELATIVE organization may be opened for access with either SEQUENTIAL, RANDOM, or DYNAMIC access mode, as specified in the ACCESS MODE clause in the program's FILE-CONTROL entry. Specification of a RELATIVE KEY is required for RANDOM and DYNAMIC access and optional for SEQUENTIAL access.

OPEN INPUT, with SEQUENTIAL access, sets the current record pointer to the first record of the file. If no records exist for the file, the pointer is set so that the AT END condition will exist on the next (Format 1) READ statement.

OPEN INPUT, with RANDOM or DYNAMIC access, sets the current record pointer to the first record of the file based on the existence of a valid RELATIVE KEY. If no records exist for the file with the specified key value, an INVALID KEY condition will exist on the next (Format 2) READ statement and the processing will be altered accordingly.

OPEN I-O assumes the existence of the named file and cannot be used if the file is being created by the program.

OPEN OUTPUT makes available a record area for development of one record, and creates a null file (no data records). After the record fields have been defined, the record will be transmitted to the assigned output device by the WRITE statement. An existing file which has the same name will be overwritten by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a DISK file; it permits use of the REWRITE statement to modify records which have been accessed by a READ statement.

The LOCKING IS {EXCLUSIVE | MANUAL | AUTOMATIC} clause is optional within the syntax of the SELECT clause and the OPEN statement. AUTOMATIC is the default behavior of the SELECT clause and the OPEN I-O and OPEN OUTPUT modes for RELATIVE file organization. For more details about locking, see Chapter 17, "File and Record LOCKING."

Failure to precede file reading or writing (in terms of time sequence) by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. If a file has been closed using the WITH LOCK option, it cannot be opened during the remainder of the run.

Note

While the Microsoft COBOL Compiler recognizes and checks the full language tape-handling syntax, it does not support tape-handling commands during program execution.

Example

```
OPEN INPUT INV-MSTR-FILE,
   OUTPUT INV-REPORT-FILE.
```

12.4.4 READ Statement

Purpose

Makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data-item, if one was specified.

Format

The general formats for RELATIVE file organization are:

```
READ file-name [ NEXT ] RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
     [ ; AT END imperative-statement ]

READ file-name RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
     [ ; INVALID KEY imperative-statement ]
```

Remarks

Format 1 must be used for all files that are being processed in SEQUENTIAL access mode.

If the file's declared mode of access is DYNAMIC, the NEXT phrase must be present to achieve SEQUENTIAL access.

The AT END phrase, if given, is executed when the logical end-of-file condition exists, or, if not given, the appropriate DECLARATIVES error section, if available, is given control.

The INTO option permits the user to copy a data record into a predefined data field for comparison with the contents in the file's record area. The predefined data field must not be defined in the FILE SECTION.

Then, using an IF statement to test for a type-field in each record, the user will be able to distinguish between the types of records that are possible, and determine exactly which type is currently available.

Format 2 must be used when RANDOM access has been defined in the ACCESS MODE clause. This format also applies if the method of record access is DYNAMIC.

If a RELATIVE KEY is defined in the FILE-CONTROL entry, successful execution of a Format 1 READ statement updates the contents of the RELATIVE KEY item (data-name-1) so as to contain the record number of the record retrieved.

For a Format 2 READ, the record that is retrieved is the one whose relative record number is pre-stored in the RELATIVE KEY item. If no such record exists, however, the INVALID KEY condition arises, and is handled by

1. the imperative statements given in the INVALID KEY portion of the READ, or
2. an associated DECLARATIVES Region

The WITH LOCK and WAIT options that have been added to the syntax of the READ and START statements for Indexed and Relative file I-O are optional. For more details about locking, see Chapter 17, "File and Record LOCKING."

Examples

```
READ INV-REC-FILE NEXT RECORD
   INTO REC-COUNT
   AT END PERFORM P300.

READ INV-REC-FILE RECORD INTO REC-COUNT
   INVALID KEY DISPLAY "REC NOT FOUND".
```

12.4.5 REWRITE Statement

Purpose

Replaces a logical record in the file.

Format

The format of the REWRITE statement is the same for RELATIVE file organization as it is for INDEXED file organization:

```
REWRITE record-name [ FROM identifier ] [ ; INVALID KEY imperative-statement ]
```

Remarks

For a file with SEQUENTIAL access, the immediately previous action would have been a successful READ; the record made available by this READ is replaced in the file by executing REWRITE. If the previous READ was unsuccessful, a runtime error will terminate execution. Therefore, no INVALID KEY phrase is allowed for SEQUENTIAL access.

For a file with DYNAMIC or RANDOM access declared, the record that is replaced by executing REWRITE is the one whose ordinal number is preset in the RELATIVE KEY item. If no such item exists, the INVALID KEY condition arises.

Example

```
REWRITE PR-REC FROM INV-REC
   INVALID KEY PERFORM P300.
```

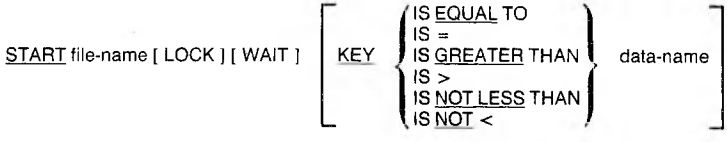
12.4.6 START Statement

Purpose

Enables a Relative file to be positioned for reading at a specified key value. This statement is allowed only for files whose access mode is defined as SEQUENTIAL or DYNAMIC.

Format

The format of the START statement is the same for RELATIVE file organization as it is for INDEXED file organization:



Remarks

Data-name may only be that of the previously declared RELATIVE KEY item, and the number of the relative record must be stored in it before START is executed. When this statement is executed, the associated file must be currently open in INPUT or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no such relative record is found, the imperative statements in the INVALID KEY phrase are executed, or an appropriate DECLARATIVES Region error-handling section is executed.

The WITH LOCK and WAIT options that have been added to the syntax of the READ and START statements for Indexed and Relative file I-O are optional. For more details about locking, see Chapter 17, "File and Record LOCKING."

Example

```
START INV-REC-FILE
   KEY IS EQUAL TO QTY-RECEIVED
   INVALID KEY DISPLAY "KEY NOT FOUND".
```

12.4.7 UNLOCK Statement

Purpose

Unlocks files formerly locked with a READ or START statement using the WITH LOCK option, or releases the last automatically locked record.

Format

The format of the UNLOCK statement is the same for RELATIVE file organization as it is for INDEXED file organization:

```
UNLOCK file-name
```

Remarks

File-name specifies the file of current reference for either a READ or a START statement. The UNLOCK statement must appear in the same run unit as the READ or a START statement whose operand is the object of its action.

Example

```
UNLOCK MSTR-ACCT-PAYABLE-FILE.
```

12.4.8 WRITE Statement

Purpose

Releases a logical record from the record processing buffer and directs the operating system to transfer the record to the designated output device; this only applies to files in OPEN OUTPUT or OPEN INPUT-OUTPUT mode.

Format

The format of the WRITE statement is the same for RELATIVE file organization as it is for INDEXED file organization:

```
WRITE record-name [ FROM identifier ] [ ; INVALID KEY imperative-statement ]
```

Remarks

If access mode is SEQUENTIAL, completion of a WRITE statement causes the relative record number of the record just output to be placed in the RELATIVE KEY item.

If access mode is RANDOM or DYNAMIC, the user must preset the value of the RELATIVE KEY item in order to assign the record an ordinal (relative) number. The INVALID KEY condition arises if there already exists a record having the specified ordinal number, or if the disk space is exceeded.

Example

```
WRITE INV-REC FROM PR-REC
   INVALID KEY PERFORM P800-NOT-FOUND.
```

Chapter 13
SORT/MERGE Facility

The SORT/MERGE Facility of Microsoft COBOL enables you to order one or more files or to combine two or more identically ordered files during program execution. These files are ordered according to a set of keys that you specify within each record.

This chapter describes the command syntax for the SORT and MERGE statements. These descriptions are preceded by the following paragraphs about related source program syntax.

13.1 Syntax Considerations

The SORT and MERGE statements use information provided by FILE-CONTROL, File Description, and I-O-CONTROL paragraphs.

13.1.1 FILE-CONTROL Entry

The FILE-CONTROL entry names a sort or merge file, using the following syntax:

```
SELECT file-name ASSIGN TO DISK

    [ SORT STATUS IS identifier]
```

Each sort or merge file described in the DATA DIVISION must be named once and only once as a file-name in the FILE-CONTROL paragraph. Each sort or merge file specified in the entry must have a SORT/MERGE file description (SD) entry in the FILE SECTION of the DATA DIVISION. In a SORT FILE-CONTROL entry, only the ASSIGN and STATUS clauses are permitted to follow file-name in the FILE-CONTROL paragraph. The identifier must be defined in the WORKING-STORAGE SECTION.

After the execution of any SORT or MERGE statement for file-name, the value of the identifier will be set to a two-digit status code. (See Section 13.2, "Sort File Status Reporting," for a list of possible error code values.) Therefore, the identifier should be described in the DATA DIVISION as a two-character alphanumeric field with USAGE DISPLAY.

13.1.2 Sort File Description Entry (SORT/MERGE)

A sort file description (SD) gives information about the file to be sorted or merged using the following syntax:

```
[ SD file name

      [ ; RECORD CONTAINS [ integer-1 TO ] integer-2 CHARACTERS ]

      [ ; DATA { RECORD IS / RECORDS ARE } data-name-1 [ , data-name-2 ] ... ]

      [ ; VALUE OF FILE-ID IS { data-name-1 / literal-1 } . ]

    { record-description entry }  ... ] ...
```

The clauses following the name of the file are optional and they may appear in any order. The RECORD clause, DATA RECORD(S) clause, and VALUE OF FILE-ID clause are described in Chapter 6, "DATA DIVISION." One or more record description entries must follow the SORT/MERGE file description entry. However, no INPUT-OUTPUT statements may be executed for this file.

13.1.3 I-O-CONTROL Paragraph

The I-O-CONTROL paragraph specifies the memory area to be shared by different files using the following syntax:

```
[ SAME [ RECORD / SORT / SORT-MERGE ] AREA FOR file-name-3 { , file-name-4 } ... ] ...
```

The I-O-CONTROL paragraph is optional. The SAME SORT AREA and SAME SORT-MERGE AREA clauses are equivalent.

The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same

time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened file whose file-name appears in this SAME RECORD AREA clause.

If the SAME SORT AREA or SAME SORT-MERGE AREA clause is used, at least one of the files must represent a SORT or MERGE file. Files that do not represent sort or merge files can be named in the clause, and files named in a SAME SORT AREA or SAME SORT-MERGE AREA clause can also be named in SAME RECORD AREA clause(s).

A file-name must not appear in more than one SAME RECORD AREA clause. Neither may a file-name that represents a sort or merge file appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.

If a file-name that does not represent a sort or merge file appears in a SAME AREA clause and in one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, all files named in that SAME AREA clause must be named in that SAME SORT AREA or SAME SORT-MERGE AREA clause(s).

The files referred to in the SAME RECORD AREA, SAME SORT AREA, or SAME SORT-MERGE AREA clause do not need to have the same organization or access.

13.2 Sort File Status Reporting

Microsoft COBOL includes a special extension to the COBOL standard, the SORT STATUS register.

The SORT STATUS register makes error detection possible. At the end of a program run, the SORT STATUS register contains a code for any errors encountered or "00" if no errors occurred.

The SORT STATUS IS phrase specifies a data-item in which the SORT facility can place a status code.

The WORKING-STORAGE SECTION description of the status data-item should specify a two-character alphanumeric field with USAGE DISPLAY.

Consistent with other error handling in COBOL, if no SORT STATUS item is declared and an error occurs, MS-COBOL will report the error and terminate execution.

The MS-COBOL runtime errors that can relate to SORT are:

```
"Need more memory"
"Object code error"
"Illegal release"
"Illegal return"
"Read beyond eof"
"Sort error XX"
```

XX can be one of the following sort file error codes:

```
00  Successful completion
20  Unknown error in Sort
70  Disk exhausted during Sort
```

Error codes 71-76 refer to system calls for intermediate files:

```
71  Error during Open call
72  Error during Close call
73  Error during Write call
74  Error during Read call
75  Error during Create call
76  Error during Delete call
```

The program will terminate abnormally, giving the message "Illegal release" or "Illegal return," if the runtime tries to execute a RELEASE or RETURN statement when no SORT is active.

13.3 SORT Statement

Purpose

Takes a sequence of records from a USING file or INPUT PROCEDURE, creates a SORT file, SORTs this file according to a set of specified keys, and then transfers the ordered records one at a time to a GIVING file or OUTPUT PROCEDURE.

Format

The format for the SORT statement is:

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [ , data-name-2 ] ...
                    { DESCENDING }

    [ ON { ASCENDING  } KEY data-name-3 [ , data-name-4 ] ... ] ...
         { DESCENDING }

    [ COLLATING SEQUENCE IS alphabet-name ].

    { INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
    {                                    { THRU    }                  }
    { USING file-name-2 [ , file-name-3 ] ...                         }

    { OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
    {                                      { THRU    }                  }
    { GIVING file-name-4                                                }
```

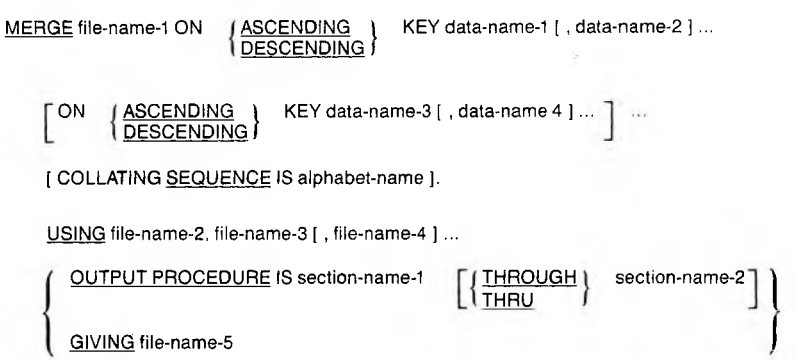
13.4 MERGE Statement

Purpose

Combines two or more identically sequenced files on a set of specified keys. The merged records are made available to an OUTPUT PROCEDURE or are output to a file.

Format

The format for the MERGE statement is:



Remarks for SORT and MERGE Operations

File-name-1 must be described in a SORT/MERGE file description entry (SD) in the DATA DIVISION. Section-name-1 represents the name of an INPUT PROCEDURE which can be declared for SORT but not for MERGE. Section-name-3 represents the name of an OUTPUT PROCEDURE.

File-name-2, file-name-3, and file-name-4 must be described in a file description entry (FD) in the DATA DIVISION. These files may have any type of organization, but ACCESS MODE must be SEQUENTIAL. If file-name-4 has INDEXED organization, the records must have been sorted according to increasing values of the RECORD KEY, or an error will occur during record output.

Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names and are subject to the following rules:

1. The data-items identified by KEY data-names must be described in records associated with file-name-1.
2. KEY data-names may be qualified.
3. If file-name-1 has more than one record description, the data-items identified by KEY data-names only need to be described in one of the record descriptions.
4. None of the data-items identified by KEY data-names can be described by an entry that contains an OCCURS clause or is subordinate to an entry which contains an OCCURS clause.
5. A maximum of 12 KEY data-names may be specified. Each KEY data-item may be from 1 to 255 characters in length.

The words THRU and THROUGH are equivalent. SORT statements may not appear in the DECLARATIVES portion of the PROCEDURE DIVISION or in an INPUT or OUTPUT PROCEDURE associated with a SORT or MERGE statement.

The COLLATING SEQUENCE phrase applies to the comparison and collating of nonnumeric key data-items. The collating sequence specified in the COLLATING SEQUENCE phrase overrides any sequence specified in the PROGRAM COLLATING SEQUENCE clause (see Section 5.2.3, "SPECIAL-NAMES Paragraph," for details).

13.5 Sorting and Merging Sequence

The data-names following the word KEY are listed from left to right in order of decreasing significance. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, etc.

When ASCENDING is specified, the sorted or merged sequence will be from the lowest value of the contents of the key data-items to the highest value. When DESCENDING is specified, the sorted or merged sequence will be from the highest value

to the lowest value Comparisons are made according to the rules for the comparison of operands in a relation condition.

For a sort file, if the KEY data-items are the same for two or more records, the sorted sequence will be the sequence in which the records were RELEASEd to the SORT by an INPUT PROCEDURE, or the sequence in which the records existed in a USING file.

13.5.1 INPUT PROCEDURE and USING Phrase

AN INPUT PROCEDURE can be used with a SORT statement, but not with a MERGE statement.

If an INPUT PROCEDURE is used, control is passed to it at the beginning of the SORT statement execution. The INPUT PROCEDURE then provides the SORT mechanism (or sorter) with records by performing a RELEASE operation for each record. Once all of the records to be sorted have been given to the sorter, the INPUT PROCEDURE exits by simply reaching the end of the section.

Control is returned to the sorter at this time, and the actual SORT operation begins.

If the USING phrase is specified, all the records in file-name-2 and file-name-3 are transferred automatically to file-name-1. File-name-2 and file-name-3 must not be open at the time of SORT or MERGE statement execution. The SORT or MERGE statement automatically initiates the processing, makes available the logical records, and terminates the processing of file-name-2 and file-name-3.

These functions allow any associated USE procedures to be executed. The terminating function for all files is performed as if a CLOSE statement without optional phrases had been executed for the file. The SORT or MERGE statement also automatically moves records from the area of file-name-2 and file-name-3 to the area for file-name-1 and releases the records to the initial input phase of the SORT or MERGE operation.

13.5.2 OUTPUT PROCEDURE and GIVING Phrase

If an OUTPUT PROCEDURE is used, control is passed to it at the final phase of the SORT or MERGE statement execution. The OUTPUT PROCEDURE receives sorted records from the sorter by performing a RETURN operation for each record. Once all of the sorted records have been given to the OUTPUT PROCEDURE, the OUTPUT PROCEDURE exits by simply reaching the end of the section.

Control is returned to the sorter at this time, the SORT operation terminates, and control returns to the statement following the SORT or MERGE statement.

If the GIVING phrase is specified, all the sorted or merged records in file-name-1 are automatically written to file-name-4 as the implied OUTPUT PROCEDURE for the respective SORT or MERGE statement. File-name-4 must not be open at the time of the SORT or MERGE statement execution. The SORT or MERGE statement automatically initiates the processing of, releases the logical records to, and terminates the processing of file-name-4.

These functions allow any associated USE procedures to be executed. The terminating function is performed as if a CLOSE statement without optional phrases had been executed for the file. The SORT or MERGE statement also automatically returns the sorted or merged records from the final phases of the SORT or MERGE operation, and then moves the records from the area for file-name-1 to the area for file-name-4.

13.6 Restrictions

The restrictions on the procedural statements within the INPUT and OUTPUT PROCEDURES are as follows:

1. The INPUT and OUTPUT PROCEDURES must not contain any SORT or MERGE statements.

2. The procedures must not contain any explicit transfers of control to points outside the procedures; ALTER, GO TO, and PERFORM statements in these procedures are not permitted to refer to procedure-names outside the INPUT and OUTPUT PROCEDURES. MS-COBOL statements that will cause an implied transfer of control to DECLARATIVES are allowed.

3. The remainder of the PROCEDURE DIVISION must not contain any transfers of control to points inside the INPUT and OUTPUT PROCEDURES; ALTER, GO TO, and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to procedure-names within the INPUT and OUTPUT PROCEDURES.

Segmentation can be applied to programs containing the SORT or MERGE statement; however, the following restrictions apply:

1. If a SORT or MERGE statement appears in a section that is not in an independent segment, any INPUT PROCEDURES or OUTPUT PROCEDURES referenced by that statement must appear within fixed segments or be contained in a single independent segment.

2. If a SORT or MERGE statement appears in an independent segment, any INPUT PROCEDURES or OUTPUT PROCEDURES referenced by that statement must be contained within fixed segments or be within the same independent segment as the SORT or MERGE statement.

13.7 RELEASE Statement

Purpose

Transfers records to the initial input phase of a SORT operation.

Format

The format for the RELEASE statement is:

```
RELEASE record-name [ FROM identifier ]
```

Remarks

A RELEASE statement may only be used within the range of an INPUT PROCEDURE associated with a SORT statement. Record-name must be the name of a record in the associated SORT/MERGE file description entry and may be qualified.

The execution of a RELEASE statement releases the record named by record-name to the initial input phase of a SORT operation.

If the FROM phrase is used, the contents of the data area referred to by the identifier-name are moved to record-name, and the contents of record-name are released to the SORT file.

Moving takes place according to the rules specified for the MOVE statement. After the RELEASE statement is executed, the information in the record area is no longer available, but the information in the data area associated with the identifier is available.

13.8 RETURN Statement

Purpose

Obtains either sorted records from the final phase of a SORT operation or merged records during a MERGE operation.

Format

The format for the RETURN statement is:

```
RETURN file-name RECORD [ INTO identifier ]
  ; AT END imperative-statement
```

Remarks

File-name must be described by a SORT/MERGE file description entry in the DATA DIVISION. A RETURN statement may only be used within an OUTPUT PROCEDURE associated with a SORT or MERGE statement for file-name.

When the logical records of a file are described with more than one record description, these records automatically share the same storage area. This is equivalent to an implicit redefinition of the area.

The contents of any data-items lying beyond the range of the current data record are undefined at the end of RETURN statement execution. RETURN statement execution causes the next record to be made available for processing in the record areas associated with the sort or merge file. The next record will be made available in the order specified by the keys listed in the SORT or MERGE statement.

If the INTO phrase is specified, the current record is moved from the input area to the area specified by the identifier according to the rules for the MOVE statement. The implied MOVE does not occur if there is an AT END condition. Any subscripting or indexing associated with the identifier is evaluated after the record has been returned and immediately before it is moved to the data-item.

When the INTO phrase is used, the data is available in both the input record area and the data area associated with the identifier.

If no next logical record exists for the file during RETURN statement execution, the AT END condition occurs. The contents of the record areas associated with the file when the AT END condition occurs are undefined. After the execution of the imperative-statement in the AT END phrase, no RETURN statement may be executed as part of the current OUTPUT PROCEDURE.

13.9 Examples

The sample programs that follow illustrate the command language for the SORT operation. The second example program, SAMPL2, illustrates the use of a Microsoft extension, the SORT STATUS clause, which is used for SORT file status reporting.

Example 1:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID. SAMPL1.
 3
 4       ENVIRONMENT DIVISION.
 5       INPUT-OUTPUT SECTION.
 6       FILE-CONTROL.
 7           SELECT INPUT-FILE ASSIGN TO DISK
 8                  ORGANIZATION IS LINE SEQUENTIAL.
 9           SELECT OUTPUT-FILE ASSIGN TO DISK
10                  ORGANIZATION IS LINE SEQUENTIAL.
11           SELECT SORT-FILE ASSIGN TO DISK.
12
13       DATA DIVISION.
14       FILE SECTION.
15       FD INPUT-FILE
16             LABEL RECORDS ARE STANDARD
17             VALUE OF FILE-ID IS "SAMPLE.IN".
18       01   INPUT-RECORD.
19            04   IN-DATE.
20                     08   IN-MONTH     PIC 99.
21                     08   IN-DAY       PIC 99.
22                     08   IN-YEAR      PIC 99.
23            04   IN-TRANSACTION-CODE PIC XXX.
24            04   IN-ACCOUNT-NUMBER   PIC 99999.
25            04   IN-REFERENCE        PIC X(9).
26            04   IN-AMOUNT    PIC S9(7)V99
27                     SIGN IS LEADING SEPARATE.
28       FD   OUTPUT-FILE
29               LABEL RECORDS ARE STANDARD
30               VALUE OF FILE-ID IS "SAMPLE.OUT".
31       01   OUTPUT-RECORD.
32            04   OUT-DATE.
33                     08   OUT-MONTH    PIC 99.
34                     08   OUT-DAY      PIC 99.
35                     08   OUT-YEAR     PIC 99.
36            04   OUT-TRANSACTION-CODE     PIC XXX.
37            04   OUT-ACCOUNT-NUMBER  PIC 99999.
```

```
38              04  OUT-REFERENCE          PIC X(9).
39              04  OUT-AMOUNT             PIC S9(7)V99
40                      SIGN IS LEADING SEPARATE.
41          SD  SORT-FILE.
42          01  SORT-RECORD.
43              04  SORT-DATE.
44                      08  SORT-MONTH  PIC 99.
45                      08  SORT-DAY    PIC 99.
46                      08  SORT-YEAR   PIC 99.
47              04  SORT-TRANSACTION-CODE   PIC XXX.
48              04  SORT-ACCOUNT-NUMBER PIC 99999.
49              04  SORT-REFERENCE      PIC X(9).
50              04  SORT-AMOUNT         PIC S9(7)V99
51                      SIGN IS LEADING SEPARATE.
52          WORKING-STORAGE SECTION.
53
54          PROCEDURE DIVISION.
55          BEGIN-HERE.
56                  SORT SORT-FILE
57                       ON ASCENDING KEY
58                           SORT-ACCOUNT-NUMBER
59                       USING INPUT-FILE
60                       GIVING OUTPUT-FILE.
61                  STOP RUN.
```

Input file (SAMPLE.IN):

```
11258020200077R00801337+000210000
11288010300224P00800988-000049999
12018010100011P00800998-001523141
12018020300077R00801348+000020176
12038020100145R00801359+000522200
12048042900224G00800721+000009669
12048010300011P00801007-000002774
12058020100077R00801363-000089160
```

Output file (SAMPLE.OUT):

```
12018010100011P00800998-001523141
12048010300011P00801007-000002774
11258020200077R00801337+000210000
12018020300077R00801348+000020176
12058020100077R00801363-000089160
12038020100145R00801359+000522200
11288010300224P00800988-000049999
12048042900224G00800721+000009669
          -----
            ^
            |
            |
            +----SORT KEY (SORT-ACCOUNT-NUMBER)
```

The following paragraphs highlight those portions of the program SAMPL1 which illustrate the use of the SORT statement.

1. Program lines 41 through 51

   The SORT/MERGE file description includes descriptions of the records which will be found in each file to be sorted, including data-item order and size.

2. Program lines 56 through 60

   In the SORT statement, the programmer specifies which record field will be used as the KEY for arranging the records in sequence.

   The programmer also specifies where the input will come from (a USING file) and where the output will go (a GIVING file).

3. Program line 11

   The SELECT file-name ASSIGN TO DISK statement must be included in the ENVIRONMENT DIVISION.

Example 2:

```
 1       IDENTIFICATION DIVISION.
 2       PROGRAM-ID.              SAMPL2.
 3
 4       ENVIRONMENT DIVISION.
 5       INPUT-OUTPUT SECTION.
 6       FILE-CONTROL.
 7           SELECT INPUT-FILE ASSIGN TO DISK
 8               ORGANIZATION IS LINE SEQUENTIAL.
 9           SELECT OUTPUT-FILE ASSIGN TO DISK
10               ORGANIZATION IS LINE SEQUENTIAL.
11           SELECT SORT-FILE ASSIGN TO DISK
12               SORT STATUS IS COMPLETION-STATUS.
13
14       DATA DIVISION.
15       FILE SECTION.
16       FD  INPUT-FILE
17           LABEL RECORDS ARE STANDARD
18           VALUE OF FILE-ID IS "SAMPLE.IN".
19       01  INPUT-RECORD.
20           04  IN-DATE.
21                   08  IN-MONTH     PIC 99.
22                   08  IN-DAY       PIC 99.
23                   08  IN-YEAR      PIC 99.
24           04  IN-TRANSACTION-CODE      PIC XXX.
25           04  IN-ACCOUNT-NUMBER        PIC 99999.
26           04  IN-REFERENCE     PIC X(9).
27           04  IN-AMOUNT        PIC S9(7)V99
28                   SIGN IS LEADING SEPARATE.
29       FD  OUTPUT-FILE
30           LABEL RECORDS ARE STANDARD
31           VALUE OF FILE-ID IS "SAMPLE.OUT".
32       01  OUTPUT-RECORD.
33           04  OUT-DATE.
34                   08  OUT-MONTH    PIC 99.
35                   08  OUT-DAY      PIC 99.
36                   08  OUT-YEAR     PIC 99.
37           04  OUT-TRANSACTION-CODE     PIC XXX.
38           04  OUT-ACCOUNT-NUMBER       PIC 99999.
39           04  OUT-REFERENCE        PIC X(9).
40           04  OUT-AMOUNT           PIC S9(7)V99
41                   SIGN IS LEADING SEPARATE.
42       SD  SORT-FILE
43           VALUE OF FILE-ID IS "SORTWORK".
44       01  SORT-RECORD.
45           04  SORT-DATE.
46                   08  SORT-MONTH      PIC 99.
47                   08  SORT-DAY        PIC 99.
```

```
48                  08  SORT-YEAR       PIC 99.
49              04  SORT-TRANSACTION-CODE PIC XXX.
50              04  SORT-ACCOUNT-NUMBER   PIC 99999.
51              04  SORT-REFERENCE        PIC X(9).
52              04  SORT-AMOUNT         PIC S9(7)V99
53                      SIGN IS LEADING SEPARATE.
54
55          WORKING-STORAGE SECTION.
56
57          01  SWITCHES.
58              05  END-OF-INPUT-SW PIC X  VALUE "N".
59                  88  END-OF-INPUT       VALUE "Y".
60              05  END-OF-OUTPUT-SW PIC X VALUE "N".
61                  88  END-OF-OUTPUT      VALUE "Y".
62
63          01  WORK-AREAS.
64              05  COMPLETION-STATUS   PIC XX.
65              05  RECORD-COUNT PIC 999 VALUE ZERO.
66              05  LAST-TRANSACTION-CODE    PIC XXX
67                                  VALUE HIGH-VALUES.
68              05  DEBIT-TOTAL PIC S9(9)V99
69                      SIGN IS LEADING SEPARATE.
70              05  CREDIT-TOTAL         PIC S9(9)V99
71                      SIGN IS LEADING SEPARATE.
72              05  DEBIT-PRINT  PIC $$$,$$$,$$9.99-.
73              05  CREDIT-PRINT PIC $$$,$$$,$$9.99-.
74              05  SPACE-FILLER PIC X(20) VALUE "  ".
75
76          PROCEDURE DIVISION.
77
78          MAIN-PROCESSING SECTION.
79
80          100-BEGIN-HERE.
81              DISPLAY
82                 "CONTROL TOTALS FOR REPORT FILE".
83              DISPLAY SPACE.
84              OPEN INPUT INPUT-FILE,
85                   OUTPUT OUTPUT-FILE.
86
87              SORT SORT-FILE
88                  ON ASCENDING KEY
89                      SORT-TRANSACTION-CODE
90                  ON DESCENDING KEY
91                      SORT-YEAR SORT-MONTH SORT-DAY
92                  INPUT PROCEDURE IS
93                      INPUT-TO-SORT
94                  OUTPUT PROCEDURE IS
95                      OUTPUT-FROM-SORT.
96
```

```
 97          IF COMPLETION-STATUS IS EQUAL TO ZERO
 98              DISPLAY "SUCCESSFULLY SORTED ",
 99                      RECORD-COUNT, " RECORDS"
100          ELSE
101              DISPLAY "SORT ERROR, STATUS IS ",
102                      COMPLETION-STATUS.
103
104          CLOSE INPUT-FILE, OUTPUT-FILE.
105          STOP RUN.
106
107        *
108        *  ***** START OF INPUT PROCEDURE *****
109        *
110
111      INPUT-TO-SORT   SECTION.
112
113      200-PROCESS-INPUT.
114          PERFORM 300-PROCESS-INPUT-FILE
115              UNTIL END-OF-INPUT.
116
117          GO TO 499-END-OF-INPUT-PROCEDURE.
118
119
120        *
121        *  ***** INPUT PROCEDURE ROUTINES *****
122        *
123
124      300-PROCESS-INPUT-FILE.
125           READ INPUT-FILE
126             AT END MOVE "Y" TO
127                END-OF-INPUT-SW.
128           IF NOT END-OF-INPUT
129             PERFORM 400-PROCESS-INPUT-RECORD.
130
131      400-PROCESS-INPUT-RECORD.
132           IF  IN-TRANSACTION-CODE
133                   IS LESS THAN "300"
134               ADD 1 TO RECORD-COUNT
135               RELEASE SORT-RECORD
136                   FROM INPUT-RECORD.
137
138      499-END-OF-INPUT-PROCEDURE.
139           EXIT.
140
141        *
142        *  ***** START OF OUTPUT PROCEDURE *****
143        *
144
145      OUTPUT-FROM-SORT   SECTION.
```

```
146
147       500-PROCESS-OUTPUT.
148           PERFORM 600-PROCESS-OUTPUT-FILE
149               UNTIL END-OF-OUTPUT.
150
151           GO TO 899-END-OF-OUTPUT-PROCEDURE.
152
153           *
154           *  ***** OUTPUT PROCEDURE ROUTINES *****
155           *
156
157       600-PROCESS-OUTPUT-FILE.
158           RETURN SORT-FILE INTO OUTPUT-RECORD
159               AT END
160                   MOVE "Y" TO END-OF-OUTPUT-SW
161                   PERFORM 800-CONTROL-TOTALS.
162
163           IF NOT END-OF-OUTPUT
164             PERFORM 700-PROCESS-OUTPUT-RECORD.
165
166       700-PROCESS-OUTPUT-RECORD.
167           IF  SORT-TRANSACTION-CODE
168               IS NOT EQUAL TO
169                   LAST-TRANSACTION-CODE
170               PERFORM 800-CONTROL-TOTALS.
171
172           IF  SORT-AMOUNT IS GREATER THAN ZERO
173               ADD SORT-AMOUNT TO DEBIT-TOTAL
174           ELSE
175               SUBTRACT SORT-AMOUNT
176                   FROM CREDIT-TOTAL.
177
178           WRITE OUTPUT-RECORD.
179
180       800-CONTROL-TOTALS.
181           IF  LAST-TRANSACTION-CODE
182                 IS NOT EQUAL TO HIGH-VALUES
183               MOVE DEBIT-TOTAL TO DEBIT-PRINT
184               MOVE CREDIT-TOTAL TO CREDIT-PRINT
185               DISPLAY "TRANSACTION CODE ",
186                  LAST-TRANSACTION-CODE,
187                  "    DEBITS " DEBIT-PRINT
188               DISPLAY SPACE-FILLER,
189                  "    CREDITS" CREDIT-PRINT.
190
191           MOVE SORT-TRANSACTION-CODE
192               TO LAST-TRANSACTION-CODE.
193           MOVE 0 TO DEBIT-TOTAL.
194           MOVE 0 TO CREDIT-TOTAL.
```

```
195
196       899-END-OF-OUTPUT-PROCEDURE.
197            EXIT.
198

  No errors or warnings

  Data area size=       1042
  Code area size=        822
```

Input File (SAMPLE.IN):

```
11258020200077R00801337+000210000
11288010300224P00800988-000049999
12018010100011P00800998-001523141
12018020300077R00801348+000020176
12038020100145R00801359+000522200
12048042900224G00800721+000009669
12048010300011P00801007-000002774
12058020100077R00801363-000089160
```

Output of program SAMPL2:

a) Output File (SAMPLE.OUT)

```
12018010100011P00800998-001523141
12048010300011P00801007-000002774
11288010300224P00800988-000049999
12058020100077R00801363-000089160
12038020100145R00801359+000522200
11258020200077R00801337+000210000
12018020300077R00801348+000020176
------
^      ---
|      ^
|      |
|      +-------Ascending major key
|              (SORT-TRANSACTION-CODE)
|
+------------Descending minor key (SORT-DATE)
```

b) Displayed Report

CONTROL TOTALS FOR REPORT FILE

```
TRANSACTION CODE 101     DEBITS          $0.00
                         CREDITS    $15,231.41
TRANSACTION CODE 103     DEBITS          $0.00
                         CREDITS       $527.73
TRANSACTION CODE 201     DEBITS       $5,222.00
                         CREDITS       $891.60
TRANSACTION CODE 202     DEBITS       $2,100.00
                         CREDITS         $0.00
TRANSACTION CODE 203     DEBITS        $201.76
                         CREDITS         $0.00
SUCCESSFULLY SORTED 007 RECORDS
```

The following paragraphs highlight those portions of program SAMPL2 that are different from program SAMPL1.

1. Program lines 42 through 53

   In addition to the descriptions of the records to be sorted, the SORT-FILE description entry now includes the optional VALUE OF FILE-ID clause which is ignored by Microsoft COBOL.

2. Program lines 87 through 95

   In SAMPL2, INPUT and OUTPUT PROCEDURES were used instead of files in USING/GIVING statements.

   Mixing of input and output selections is acceptable; that is, you may select an INPUT PROCEDURE and an output (GIVING) file, or an input (USING) file and an OUTPUT PROCEDURE; these selections are as acceptable as the selections in SAMPL1 and SAMPL2.

3. Program lines 97 through 102

   Each COBOL program that uses the SORT/MERGE Facility should define and test a SORT STATUS register so that any errors will be reported at the end of the program run. This extension to the 1974 ANSI standard will make error detection possible and will make error handling and debugging easier. Refer to Section 13.2, "Sort File Status Reporting," for a listing of the possible SORT STATUS register values.

4. Program lines 111 through 139

   The INPUT-TO-SORT SECTION was written because special user processing was desired for each input record. (The special processing in this case is the transaction code selection at line 132.)

   The program lines here specify the INPUT PROCEDURE. This section includes the RELEASE statement to transfer records to the initial phase of the SORT operation. Notice the PERFORM loop inside procedure 200-PROCESS-INPUT. Remember: The INPUT PROCEDURE is processed only once, so any necessary looping must be included in the INPUT PROCEDURE.

   The GO TO statement at the end of procedure 200-PROCESS-INPUT is present because an INPUT PROCEDURE implicitly PERFORMS the entire section specified, not just the first paragraph in the section. Without the GO TO, control would have fallen into 300-PROCESS-INPUT-FILE when the PERFORM in 200-PROCESS-INPUT was complete. The GO TO directs control to the end of INPUT-TO-SORT SECTION, where it will return to the SORT statement.

5. Program lines 145 through 197

   An OUTPUT PROCEDURE in the OUTPUT-FROM-SORT SECTION was written to make immediate use of the sort output by producing a short report.

   This is typical of an OUTPUT PROCEDURE through which sorted records are processed from SORT or MERGE operations. This section will include the RETURN statement to transfer records from the final phase of a SORT or MERGE operation. Notice the PERFORM loop inside procedure 500-PROCESS-OUTPUT. As with the INPUT PROCEDURE, the OUTPUT PROCEDURE is processed only once, so any necessary looping must be included in the OUTPUT PROCEDURE.

6. Program lines 12 and 64

   The SORT STATUS IS phrase specifies a data-item (in this case COMPLETION-STATUS) in which SORT will place a status code. The WORKING-STORAGE

description of the status data-item should specify a two-character field with USAGE DISPLAY.

Consistent with other error handling in COBOL, if no SORT STATUS item is declared and an error occurs, MS-COBOL will report the error and terminate execution.

Chapter 14

DECLARATIVES Region and USE Statement

The section and entry code that makes up the DECLARATIVES Region provides a method of including procedures that are executed not as part of the sequential coding written by the programmer, but rather when a condition occurs that cannot normally be tested by the programmer.

Although the system automatically handles checking and creation of standard labels and executes error recovery routines in the case of I-O errors, additional procedures may be specified.

Since these procedures are executed only at the time an error in reading or writing occurs, they cannot appear in the regular sequence of procedural statements. They must be written at the beginning of a PROCEDURE DIVISION.

Related procedures are preceded by a USE statement that specifies their function. A declarative section ends with the occurrence of another section-name with a USE statement or with the key words END DECLARATIVES.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period (.).

The general format is:

```
[ DECLARATIVES.

{ section-name SECTION [ segment-number ]. USE statement.

[ paragraph-name. [ sentence] ... ] ... } ...

 END DECLARATIVES. ]
```

The USE statement defines the applicability of the associated section of coding.

A USE statement, when present, must immediately follow a SECTION header in the DECLARATIVES Region of a PROCEDURE DIVISION and must be followed by a period followed by a space. The remainder of the section must consist of zero, one, or more procedural paragraphs that define the procedures to be used. The USE statement itself is never executed; rather, it defines the conditions under which the standard error or exception procedure will be executed. The general format of the USE statement is:

The words EXCEPTION and ERROR may be used interchangeably. The associated DECLARATIVES Region is executed (by the PERFORM mechanism) after the standard I-O recovery procedures for the files designated, or after the INVALID KEY or AT END condition arises on a statement lacking the INVALID KEY or AT END phrase. A given filename may not be associated with more than one DECLARATIVES Region.

Within a DECLARATIVES Region there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in a DECLARATIVES Region, except that PERFORM statements may refer to a USE statement and its procedures; but in a range specification (see Section 7.6.22, "PERFORM Statement"), if one procedure-name is in a DECLARATIVES Region, the other must be in the same DECLARATIVES Region.

An exit from a DECLARATIVES Region is inserted by the compiler following the last statement in the section. All logical program paths within the section must lead to the exit point.

Chapter 15
Segmentation

The program segmentation facility is provided to enable the execution of Microsoft COBOL programs that are larger than physical memory. When segmentation is used (that is, when any section header in the program contains a segment number), the entire PROCEDURE DIVISION must be written in sections. Each section is assigned a segment number by a section header of the form:

```
section-name SECTION [ segment-number] .
```

The general rules that follow apply to the use of segment-numbers:

1. Segment-number must be an integer with a value in the range from 0 through 99.
2. If the segment-number is omitted, it is assumed to be 0.
3. A DECLARATIVES Region must have segment-numbers less than 50.
4. All sections which have the same segment-number constitute a single program segment and must occur together in the source program, but do not need to be contiguous.
5. All segments with numbers less than 50 must occur together at the beginning of the PROCEDURE DIVISION.

Segments with numbers 0 through 49 are called fixed segments (permanent) and are always resident in memory during execution, except as redefined by the SEGMENT-LIMIT clause. If the SEGMENT-LIMIT clause has been used, the segment-numbers from the newly defined upper limit to 49 become fixed overlayable segments and may be overlaid by an independent segment at runtime.

Segments with numbers greater than 49 are called independent segments. Each independent segment is treated as a program overlay. An independent segment is in its initial state when control is passed to it for the first time during the execution of a program, and also when control is passed to that section (implicitly or explicitly) from another segment with a different segment number.

Specifically, an independent segment is in its initial state when it is reached by "falling through" the end of a fixed or different independent segment.

Segmentation causes the following restrictions on the use of the ALTER and PERFORM statements:

1. A GO TO statement in an independent segment must not be referred to by an ALTER statement in any other segment.
2. A PERFORM statement in a fixed segment may have within its range only
   a. sections and/or paragraphs wholly contained within fixed segments, or
   b. sections and/or paragraphs wholly contained in a single independent segment
3. A PERFORM statement in an independent segment may have within its range only
   a. sections and/or paragraphs wholly contained within fixed segments, or
   b. sections and/or paragraphs wholly contained within the same independent segment as the PERFORM statement

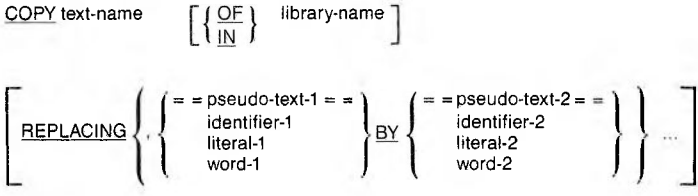
See Section 5.2.1, "OBJECT-COMPUTER Paragraph," for information about the SEGMENT-LIMIT clause.

These remarks apply to processing of any file, whether organization is SEQUENTIAL, LINE SEQUENTIAL, INDEXED, or RELATIVE.

Chapter 16
COPY Statement

The COPY statement is used to logically embed the text of a disk file (other than the source file) in the source program. The COPY statement may be used anywhere in the ENVIRONMENT, DATA, or PROCEDURE DIVISIONs.

The format of the COPY statement is:



where text-name is a disk filename in the format required by the operating system and library-name is a name whose format may vary. See the *Microsoft COBOL Compiler User's Guide* for information on the formation of file and library names.

To illustrate the use of the COPY statement, suppose that you want to copy a text file, BDEF.COB, into a host program. BDEF.COB contains the following source code:

```
05 B.
       10 B1 PIC X.
       10 B2 PIC X.
```

When the host source containing the COPY BDEF.COB code:

```
05 A.
    10 A1 PIC 9.
COPY BDEF.COB.
05 C.
    10 C1 PIC Z.
```

is compiled, the listing file will look like this:

```
43              05 A.
44                  10 A1 PIC 9.
45              COPY BDEF.COB.
46 C            05 B.
47 C                10 B1 PIC X.
48 C                10 B2 PIC X.
49              05 C.
50                  10 C1 PIC Z.
```

The COPY statement should be the last or only statement on the line. Note that the entire statement containing the COPY verb, including the terminal period (.), is replaced by the contents of file-name, so that any periods desired must be present in the copied file.

If several libraries are available during compilation, a library-name may be specified. This serves to qualify text-name and identify the library where the text will be found. See the *Microsoft COBOL Compiler User's Guide* for a description of library names allowable with your implementation.

In the REPLACING phrase, pseudo-text-1 may neither be null, nor consist only of comment lines or character spaces. Pseudo-text-2, however, may be null.

Comment lines occurring in the library text and pseudo-text-1 are interpreted as single spaces for purposes of matching. Comment lines occurring in the library text and pseudo-text-2 are copied into the host program unchanged.

Debugging lines are permitted in the library text and pseudo-text-2. Debugging lines are not permitted in pseudo-text-1; text-words appearing within a debugging line participate in the matching as if the "D" did not appear in the comment area (column 7).

Character strings within pseudo-text-1 and pseudo-text-2 may be continued, but the characters of a pseudo-text delimiter (==) must be on the same source program line.

Note

Source program lines that have been copied into the host program text are not checked for correct syntax until after the REPLACING operation has been performed. Any text placed beyond column 72 by the REPLACING operation will be accepted, and the restrictions of placement of text inside column 72 will be removed for the remainder of the current line.

Word-1 or word-2 may be any COBOL word.

If the REPLACING phrase is not specified, the library text is copied unchanged.

If the REPLACING phrase is used, the copying and replacing operations occur in the following steps:

Note

In the discussion that follows, the term "text-word" is defined as a character string or separator (not including a space) that occurs in pseudo-text or a library file.

---

1. The first operand of the REPLACING clause is compared to the text-words in the specified library, character for character, starting with the leftmost library text-word.
2. If a match exists, the text-word represented by the operand following the reserved word, BY, is placed in the source program instead of the text-word represented by psuedo-text-1, identifier-1, literal-1, or word-1.
3. If no correspondence exists during any of the comparison cycles, the library text-word that is currently the object of the comparison is placed unchanged into the

source program, and the next text-word in the library is evaluated (going left to right).

4. The comparison operation continues until the rightmost text-word in the library has either participated in a match or has gone unmatched and been placed into the source program.

Semicolons and separator commas that occur in psuedo-text-1, are considered as spaces during the comparison process, unless a semicolon or comma is the total content of the psuedo-text-1 operand. In that case, semicolons and separator commas are treated as text-words for comparison.

Examples

The examples that follow demonstrate the use of the COPY statement in an FD paragraph and in a procedure.

Suppose that you wanted to insert the following data description, and that this "text" was in a file on disk called MASTER.CPY.

```
01  MASTER-RECORD.
    05  MSTR-KEY              PIC X(10).
    05  MSTR-DESCRIPTION      PIC X(25).
    05  MSTR-AMT-ON-HAND      PIC S9(5).
    05  MSTR-WARNING-LEVEL    PIC S9(5).
```

To embed the text of MASTER.CPY in the host program, the COPY statement can be inserted at the appropriate position in the source code.

```
FD  INVENTORY-MASTER
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS "MASTER.DAT".

    COPY MASTER.CPY.
```

This COPY operation will copy the contents of MASTER.CPY into the source program under the FD, and will yield the following text in the listing file:

```
101          FD  INVENTORY-MASTER
102              LABEL RECORDS ARE STANDARD
103              VALUE OF FILE-ID IS "MASTER.DAT".
104
105              COPY MASTER.CPY.
106 C        01  MASTER-RECORD.
107 C            05  MSTR-KEY              PIC X(10).
108 C            05  MSTR-DESCRIPTION      PIC X(25).
109 C            05  MSTR-AMT-ON-HAND      PIC S9(5).
110 C            05  MSTR-WARNING-LEVEL    PIC S9(5).
```

In the next example, the COPY operation will replace existing identifiers and text strings in the library file, MAIN.CPY, with those specified with the REPLACING clause, and embed them in the host program.

MAIN.CPY is a file on disk containing this source code:

```
OPEN INPUT INVENTORY-MASTER-FILE,
     OUTPUT INVENTORY-WARNING-FILE,
     INVENTORY-REPORT-FILE.

WRITE REPORT-RECORD FROM PR-HEADER
     AFTER ADVANCING PAGE.
PERFORM P100-WRITE-REPORT
     UNTIL END-OF-FILE.

MOVE REC-COUNT TO PR-REC-COUNT.
MOVE WARNING-COUNT TO PR-WARNING-COUNT.
WRITE REPORT-RECORD
     FROM PR-TOTAL-RECORD
     AFTER ADVANCING 2 LINES.

CLOSE INVENTORY-MASTER-FILE,
     INVENTORY-WARNING-FILE,
     INVENTORY-REPORT-FILE.

STOP RUN.
```

Suppose that the text in MAIN.CPY can logically be inserted into a procedure. The COPY statement would appear in the code like this:

```
P000-MAINLINE.

    COPY MAIN.CPY
      REPLACING
        INVENTORY-MASTER-FILE
            BY TRANSACTION-FILE
        ==AFTER ADVANCING 2 LINES==
            BY ==BEFORE ADVANCING 1 LINE==
        END-OF-FILE
            BY ==REC-COUNT > 10==.
```

The listing file will look like this after compilation:

```
200          P000-MAINLINE.
201
202              COPY MAIN.CPY
203                REPLACING
204                  INVENTORY-MASTER-FILE
205                      BY TRANSACTION-FILE
206                  ==AFTER ADVANCING 2 LINES==
207                      BY
208                      ==BEFORE ADVANCING 1 LINE==
209                  END-OF-FILE
210                      BY ==REC-COUNT > 10==.
211
212 C              OPEN INPUT TRANSACTION-FILE,
213 C                   OUTPUT INVENTORY-WARNING-FILE,
214 C                   INVENTORY-REPORT-FILE.
215 C
216 C              WRITE REPORT-RECORD FROM PR-HEADER
217 C                  AFTER ADVANCING PAGE.
218 C              PERFORM P100-WRITE-REPORT
219 C                   UNTIL REC-COUNT > 10.
220 C
221 C              MOVE REC-COUNT TO PR-REC-COUNT.
222 C              MOVE WARNING-COUNT
223 C                   TO PR-WARNING-COUNT.
224 C              WRITE REPORT-RECORD
225 C                   FROM PR-TOTAL-RECORD
226 C                   BEFORE ADVANCING 1 LINE.
227 C
228 C              CLOSE TRANSACTION-FILE,
229 C                   INVENTORY-WARNING-FILE,
230 C                   INVENTORY-REPORT-FILE.
231 C
232 C              STOP RUN.
```

Chapter 17
File and Record LOCKING

A file and record LOCKING construct for multi-user/multi-tasking systems has been implemented with Microsoft COBOL. File and record LOCKING ensure protection of data files during simultaneous access by multiple processes, and give individual processes the opportunity for exclusive use of a file.

This is an extension to the full language standard, and provides for Sequential and Line Sequential file processing in one LOCKING mode (EXCLUSIVE), and Indexed and Relative file processing in three LOCKING modes (EXCLUSIVE, MANUAL, or AUTOMATIC).

The file and record LOCKING syntax is supported in these places:

1. in the FILE-CONTROL entry which supercedes the I-O statements that follow.
2. in the OPEN, READ, START, and UNLOCK statements.
3. in some implementations of MS-COBOL, additional file locking commands can be entered on the runtime command line. If your implementation supports this feature the *Microsoft COBOL Compiler User's Guide* will describe its use.

17.1 File LOCKING

File LOCKING gives the user exclusive use of the file. All attempts by another process to either read, write, or lock the file will be prevented by the operating system. If the other process is a COBOL program, an error will be generated by the COBOL runtime. File LOCKING is most frequently used for sequential processing of a file.

17.2 Record LOCKING

Record LOCKING is only implemented for Indexed and Relative files.

Selecting the AUTOMATIC option in the LOCKING clause specifies that the records will be accessed in the AUTOMATIC record LOCKING mode. The AUTOMATIC record LOCKING mode automatically locks any record that is the target of a READ statement. The same record is automatically unlocked by the next READ, WRITE, REWRITE, or CLOSE.

See Section 17.3, "Syntax Considerations," for the LOCKING clause syntax.

Selecting the MANUAL option in the LOCKING clause specifies that the records will be accessed in the MANUAL LOCKING mode. The MANUAL LOCKING mode only locks a record when the LOCK verb is present in the respective READ or START statement. In MANUAL LOCKING mode, the record is only unlocked by an UNLOCK statement or when the file is CLOSEd.

Record LOCKING gives the user exclusive use of several records at a time in the MANUAL LOCKING mode or one record at a time in the AUTOMATIC LOCKING mode. For the limitations placed on multiple record LOCKING under the MANUAL LOCKING mode, see the *Microsoft COBOL Compiler User's Guide.*

Note

Normal I-O error handling applies to this element of the construct, requiring FILE STATUS, INVALID KEY, or DECLARATIVES Region language to prevent program termination.

17.3 Syntax Considerations

The discussion that follows explains the use of the file and record LOCKING command language within either the FILE-CONTROL entry, or in the file I-O statements: OPEN, READ, START, and UNLOCK.

17.3.1 FILE-CONTROL Entry (SELECT Clause)

The four file organizations supported by Microsoft COBOL now default to specific file LOCKING and record LOCKING modes.

17.3.1.1 Sequential and Line Sequential Files

The syntax for the SELECT clause for Sequential and Line Sequential files follows:

```
SELECT [ OPTIONAL ] file-name

        ASSIGN TO  { DISK    }
                   { PRINTER }

        [ ; [ LOCKING IS ] EXCLUSIVE ]

        [ ; RESERVE integer [ AREA  ] ]
                            [ AREAS ]
        [ ; ORGANIZATION IS [ LINE ] SEQUENTIAL ]

        [ ; ACCESS MODE IS SEQUENTIAL ]

        [ ; FILE STATUS IS data-name-1 ].
```

The use of the EXCLUSIVE option in the SELECT clause for the file specifies exclusive use of the file. This mode only applies to disk files.

The EXCLUSIVE option is meant to be used with file modes and organizations where random access is not the objective.

Note

EXCLUSIVE is the default behavior of the SELECT clause for SEQUENTIAL and LINE SEQUENTIAL file organizations.

17.3.1.2 Indexed and Relative Files

The syntax for the SELECT clause for Indexed files is:

```
FILE-CONTROL.
SELECT file-name
        ASSIGN TO DISK
      [ [ LOCKING IS]  { EXCLUSIVE }  ]
                       { MANUAL    }
                       { AUTOMATIC }
      [ ; RESERVE integer  [ AREA  ]  ]
                           [ AREAS ]
        ; ORGANIZATION IS INDEXED
      [ ; ACCESS MODE IS { SEQUENTIAL } ]
                         { RANDOM     }
                         { DYNAMIC    }
```

and for Relative files is:

```
SELECT file-name
    ASSIGN TO DISK
[ [ LOCKING IS] { EXCLUSIVE } ]
                { MANUAL    }
                { AUTOMATIC }
[ ; RESERVE integer [ AREA  ] ]
                    [ AREAS ]
  ; ORGANIZATION IS RELATIVE
[ ; ACCESS MODE IS { SEQUENTIAL [ , RELATIVE KEY IS data-name-1] } ]
                   {                                             }
                   { { RANDOM  }                                 }
                   { { DYNAMIC } , RELATIVE KEY IS data-name-1   }
  [ ; FILE STATUS IS data-name-2] .
```

The use of the EXCLUSIVE option in the SELECT clause for the file specifies exclusive use of the file.

The AUTOMATIC option allows sharing of the records in the file. The AUTOMATIC option dictates that a single record will automatically be locked when accessed by a READ statement and automatically unlocked by the next READ, REWRITE, WRITE, or CLOSE statements. Only one record is locked and unlocked at a time in this context.

Note

The AUTOMATIC option is the default for Indexed and Relative files.

If the MANUAL option is specified, more than one record can be locked or unlocked at a time. The READ and START statements using the LOCK verb will lock a record. The UNLOCK statement will release all locks present in the file.

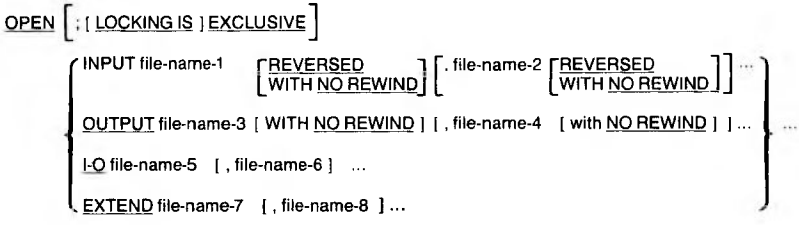
The AUTOMATIC and MANUAL options are meant to be used where random record processing of Relative and Indexed files in the OPEN I-O mode is the object of the program.

17.3.2 OPEN, READ, START, and UNLOCK Statements

The LOCKING context specified in the OPEN statement takes precedence over any subsequent file LOCKING language. For example, if the OPEN statement implies or specifies EXCLUSIVE LOCKING, subsequent LOCK, UNLOCK, and WAIT clauses for the file so opened are ignored.

17.3.2.1 OPEN Statement (Sequential and Line Sequential)

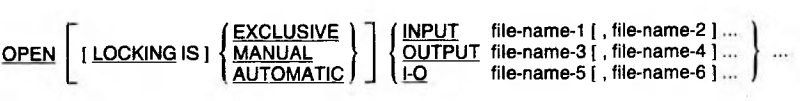
The syntax for the OPEN statement for Sequential and Line Sequential files follows:



If the LOCKING clause is not specified, the EXCLUSIVE LOCKING mode is implied by OPEN EXTEND and OPEN OUTPUT. Files OPENed with the INPUT option may only be read, and so require no file locking.

17.3.2.2 OPEN Statement (Indexed and Relative)

The syntax for the OPEN statement for Indexed and Relative files follows:



If the LOCKING clause is not specified, the EXCLUSIVE LOCKING mode is implied by OPEN OUTPUT. AUTOMATIC LOCKING mode is implied by OPEN I-O. Files OPENed with the INPUT option may only be read, and so require no file locking.

17.3.2.3 READ Statement (in MANUAL Mode)

The READ statement syntax for Indexed files is:

```
READ file-name [ NEXT ] RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
        [ ; AT END imperative-statement ]
READ file-name RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
        [ ; KEY IS data-name ]
        [ ; INVALID KEY imperative-statement ]
```

and for Relative files is:

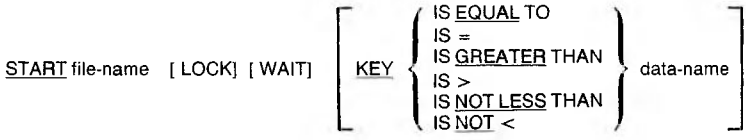
```
READ file-name [ NEXT ] RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
        [ ; AT END imperative-statement ]
READ file-name RECORD [ LOCK ] [ WAIT ] [ INTO identifier ]
        [ ; INVALID KEY imperative-statement ]
```

The LOCK and WAIT verbs are optional. If the WAIT verb is used and a locked record is encountered during runtime, the process will wait until the specified record is released, and then will execute the READ instruction.

If the WAIT verb is not specified and a locked record is encountered during runtime, the runtime executor will return an error condition.

17.3.2.4 START Statement (in AUTOMATIC and MANUAL Mode)

The syntax for the START statement for Indexed and Relative files is:



The LOCK and WAIT verbs are optional. If the WAIT verb is used and a locked record is encountered during runtime, the process will wait until the specified record is released, and then will execute the START instruction.

If the WAIT verb is not specified and a locked record is encountered during runtime, the runtime executor will return an error condition.

Unlike the READ statement which automatically locks a record in AUTOMATIC LOCKING mode, the START statement in AUTOMATIC LOCKING mode will not lock a record unless the LOCK verb is present in the statement.

17.3.2.5 UNLOCK Statement

The UNLOCK statement removes all locks from a file and, with the exception of the CLOSE statement, is the only means of unLOCKING the records of a file in the MANUAL LOCKING mode. Furthermore, the UNLOCK statement only applies within the context of the MANUAL LOCKING mode.

Appendices

Appendix A
Permissible MOVE Operands

Table A.1

Permissible MOVE Operands

	Receiving Operand In MOVE Statement					
Source Operand	**Numeric Integer**	**Numeric Non-integer**	**Numeric Edited**	**Alpha-numeric Edited**	**Alpha-numeric**	**Group**
Numeric Integer	OK	OK	OK	OK(A)	OK(A)	OK(B)
Numeric Non-integer	OK	OK	OK			OK(B)
Numeric Edited				OK	OK	OK(B)
Alpha-numeric Edited				OK	OK	OK(B)
Alpha-numeric	OK(C)	OK(C)	OK(C)	OK	OK	OK(B)
Group	OK(B)	OK(B)	OK(B)	OK(B)	OK(B)	OK(B)

The characters (A), (B), and (C) in the preceding table indicate:

(A) Source sign, if any, is ignored.

(B) If the source operand or the receiving operand is a group item, the move is considered to be a group move.

(C) Source is treated as an unsigned integer; source length may not exceed 31.

Note

No distinction is made in the compiler between alphabetic and alphanumeric. Therefore, numeric items should not be moved to alphabetic items, and vice versa.

Alignment Rules for Receiving Fields

The data type of a receiving field determines the positioning of data within it. The following rules apply:

1. If the receiving field is numeric, the data-item is aligned on the decimal point by padding with zeros or by truncation, when the data-item is too short or too long.
2. If the receiving field is numeric-edited, the data-item is aligned as in a numeric field, except that the editing may cause replacement of leading zeros and the insertion of editing characters, as specified by the receiving field's PICTURE clause.
3. If the receiving field is alphabetic, alphanumeric, or alphanumeric-edited, the data-item is aligned at the leftmost character position with SPACE FILL or truncation on the right end as required by the data transfer.

If the JUSTIFIED clause has been specified for the receiving field, these alignment rules are modified as described in the JUSTIFIED clause.

Appendix B
Nested IF Statements

A "nested IF" exists when the conjunction IF appears more than once in a single sentence.

Example

```
IF X = Y
    IF A = B
        MOVE "*" TO SWITCH
    ELSE
        MOVE "A" TO SWITCH
ELSE
    MOVE SPACE TO SWITCH.
```

The flow of the preceding example may be represented by a tree structure. Such a structure is illustrated by the figure on the following page.

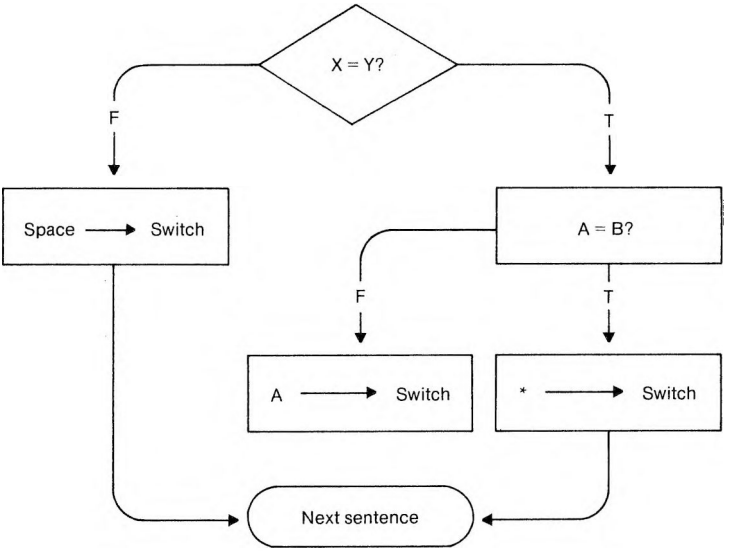


Figure B.1. Tree Structure of Nested IF Statements

Another useful way of viewing nested IF structures is based on numbering IF and ELSE verbs to show their priorities.

Example

```
IF1     X = Y
  true-action1:
    IF2     A = B
      true-action2:
        MOVE "*" TO SWITCH
    ELSE2
      false-action2:
        MOVE "A" TO SWITCH
ELSE1
  false-action1:
    MOVE SPACE TO SWITCH.
```

The preceding illustration shows that IF2 is wholly nested within the true-action side of IF1.

The number of ELSEs in a sentence need not be the same as the number of IFs; there may be fewer ELSE branches.

Example

```
IF M = 1
      IF K = 0
         GO TO M1-K0
      ELSE
         GO TO M1-KNOT0.

IF AMOUNT IS NUMERIC
   IF AMOUNT IS ZERO
       GO TO CLOSE-OUT.
```

In the latter case, the second IF could equally well have been written as AND.

Appendix C
Reserved Words

In the reserved word lists that follow, a plus sign (+) before a reserved word indicates words which are required by Microsoft COBOL for its extensions: the interactive screens feature, debug and file-locking facilities, and special data formats.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA(S)
ASCENDING
+ASCII
ASSIGN
AT
AUTHOR
+AUTO-SKIP
+AUTOMATIC

+BACKGROUND-COLOR
+BEEP
BEFORE
+BELL
BLANK
+BLINK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
+CHAIN
+CHAINING
CHARACTER(S)
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
+COL
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
+COMPUTATIONAL-0
+COMP-0
+COMPUTATIONAL-3
+COMP-3
+COMPUTATIONAL-4
+COMP-4
COMPUTE
CONFIGURATION
CONTAINS
CONTROL(S)
COPY
CORR(ESPONDING)
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT

DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DE(TAIL)
DISABLE
DISK
+DISPLAY
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
+EJECT
ELSE
EMI
+EMPTY-CHECK
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
+ERASE
ERROR
+ESCAPE
ESI
EVERY
EXCEPTION
+EXCLUSIVE
+EXHIBIT
EXIT
EXTEND

FD
FILE
FILE-CONTROL
+FILE-ID
FILLER
FINAL
FIRST
FOOTING
FOR
+FOREGROUND-COLOR
FROM

GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE(S)
HIGHLIGHT

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST(IFIED)

KEY

LABEL
+LAST
LEADING
LEFT
+LEFT-JUSTIFY
LENGTH
+LENGTH-CHECK
LESS
LIMIT(S)
+LIN
LINAGE
LINAGE-COUNTER
LINE(S)
LINE-COUNTER
+LINKAGE
LOCK
+LOCKING
LOW-VALUE(S)

+MANUAL
MEMORY
MERGE
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NO
+NO-ECHO
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC(TURE)
PLUS
POINTER
POSITION
POSITIVE
+PRINTER
PRINTING
PROCEDURE(S)
PROCEED
PROGRAM
PROGRAM-ID
+PROMPT

QUEUE
QUOTE(S)

RANDOM
RD
READ
+READY
RECEIVE
RECORD(S)
REDEFINES

REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT(S)
REPORTING
RERUN
RESERVE
RESET
RETURN
REVERSE-VIDEO
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
+RIGHT-JUSTIFY
ROUND
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE(S)
+SPACE-FILL
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING
SUB-QUEUE-1,2,3
SUBTRACT
SUM
SUPPRESS
SYMBOLIC
SYNC(HRONIZED)

TABLE
TALLYING
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THROUGH
THRU
TIME
TIMES
TO
TOP
+TRACE
TRAILING
+TRAILING-SIGN
TYPE

UNDERLINE
UNIT
+UNLOCK
UNSTRING
UNTIL
UP
+UPDATE
UPON
USAGE
USE
USING

VALUE(S)
VARYING

+WAIT
WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO((E)S)
ZERO-FILL

+
−
*
/
**
<
>
=(=)

Appendix D
ASCII Character Set

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	032	20H	SPACE
001	01H	SOH	033	21H	!
002	02H	STX	034	22H	"
003	03H	ETX	035	23H	#
004	04H	EOT	036	24H	$
005	05H	ENQ	037	25H	%
006	06H	ACK	038	26H	&
007	07H	BEL	039	27H	'
008	08H	BS	040	28H	(
009	09H	HT	041	29H	)
010	0AH	LF	042	2AH	*
011	0BH	VT	043	2BH	+
012	0CH	FF	044	2CH	,
013	0DH	CR	045	2DH	-
014	0EH	SO	046	2EH	.
015	0FH	SI	047	2FH	/
016	10H	DLE	048	30H	0
017	11H	DC1	049	31H	1
018	12H	DC2	050	32H	2
019	13H	DC3	051	33H	3
020	14H	DC4	052	34H	4
021	15H	NAK	053	35H	5
022	16H	SYN	054	36H	6
023	17H	ETB	055	37H	7
024	18H	CAN	056	38H	8
025	19H	EM	057	39H	9
026	1AH	SUB	058	3AH	:
027	1BH	ESCAPE	059	3BH	;
028	1CH	FS	060	3CH	<
029	1DH	GS	061	3DH	=
030	1EH	RS	062	3EH	>
031	1FH	US	063	3FH	?

Dec	Hex	CHR	Dec	Hex	CHR
064	40H	@	096	60H	`
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[	123	7BH	{
092	5CH	\	124	7CH	|
093	5DH	]	125	7DH	}
094	5EH	^	126	7EH	~
095	5FH	_	127	7FH	DEL

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Index