Advantage™ VISION:Inform®

ASL Reference Guide

4.0

Computer Associates®

IFASL040.PDF/D39-002-011

Contents

Chapter 1: Introduction

Chapter 2: Terminology, Syntax, and Processing

Chapter 3: Built-in Functions

Chapter 4: Procedure Statements

Chapter 5: ASL Examples

Appendix A: Relationship of ASL Statements to Host Engine Statements

Appendix B: Technical Notes

Appendix C: Flags

Appendix D: ASL Reference Summary

Index

Chapter

1

Introduction

Advanced Syntax Language (ASL) is a free-form language used to build procedures consisting of logical, arithmetic, branching, and text processing statements. It is designed with both the programmer and the end user in mind. The syntax and structure of ASL statements are similar in nature to other free-form languages such as C and Pascal. However, ASL goes beyond these languages by providing many simple yet powerful business-oriented functions.

The power of ASL is illustrated in how simple it is to change text in a field. For example, `REPLACE STRING 'ABC' IN NAME WITH 'XYZ'` does just what it says. Each time ABC is found in the field NAME, ABC is replaced with XYZ.

Field NAME contents:

Before: `'THE ABC COMPANY'`

After: `'THE XYZ COMPANY'`

Because the statements that are used to build ASL procedures are structured like sentences, the language is easy to understand, self-documenting (although comments are optional), and easy to maintain.

Operating System and Environment Support

ASL is directly supported in its native free-form syntax when using VISION:Workbench™ for DOS and VISION:Workbench for ISPF (also known as the Definition Processor).

In OS/390 (z/OS) and CMS

When using the VISION:Builder ®, Advantage VISION:Inform ®, and VISION:Two™ engines, ASL is directly supported in its native free-form syntax in the OS/390 (z/OS) and CMS environments.

In VSE

Native ASL syntax is not directly supported by the engines in the VSE environment.

For VISION:Builder and VISION:Inform users in the VSE environment, VISION:Workbench for DOS translates your native ASL syntax coding into fixed format coding during the export process. See the VISION:Workbench for DOS Setup window description in the *VISION:Workbench for DOS Reference Guide* for details on specifying target host support of native ASL.

About This Book

Note: Text that is specific to a host engine is indicated by a note.

This book contains information specific to the use and operation of ASL and assumes that you are familiar with one of the host engines, VISION:Builder, VISION:Inform, or VISION:Two.

Chapter 1, "Introduction"

Describes this book.

Chapter 2, "Terminology, Syntax, and Processing"

Starts with information about how to read the syntax of the statements and defines some common computer terminology with respect to ASL.

Chapter 3, "Built-in Functions"

Describes the built-in functions of ASL. Built-in functions are used as a part of other procedure statements. First, the syntax of the function is given, then the description of the function, followed by descriptions of each keyword phrase.

Chapter 4, "Procedure Statements"

Describes the procedure statements. Again, the syntax is displayed and defined. Examples are given to illustrate the procedure statement.

Chapter 5, "ASL Examples"

Shows examples using ASL.

Appendix A, "Relationship of ASL Statements to Host Engine Statements"

Relates the functions and procedure statements to their host engine counterparts.

Appendix B, "Technical Notes"

Contains technical information defining some of the operand entries. When necessary, this appendix will be referenced in the explanation of the operand. It also contains a reserved word list.

Appendix C, "Flags"

Lists all of the host engine flags.

Appendix D, "ASL Reference Summary"

Provides summary information and tables for ASL.

Contacting Technical Support

For technical assistance with this product, contact Computer Associates Technical Support on the Internet at esupport.ca.com. Technical support is available 24 hours a day, 7 days a week.

Chapter

2 Terminology, Syntax, and Processing

This chapter defines the terms that are used to describe ASL statements, ASL syntax, and how ASL processes.

ASL Terminology

ASL is a free-form language consisting of commands and functions. A procedure statement begins on a new line and consists of an optional label followed by a command. If the statement has a label, the label must be followed immediately (without intervening spaces) by a colon. What appears on the rest of the procedure statement after the command depends on the syntax of the command.

For example:

```
LABELX: LET FIELDA = FIELDB
```

is an actual procedure statement where:

`LABELX`	Specifies the statement label.
`LET`	Specifies the command.
`FIELDA = FIELDB`	Places the contents of field B into field A.

Syntax Terminology

The following terminology is used to describe ASL.

Term	Description
Procedure	A procedure defines an algorithm or sequence of calculations. It is composed of a series of procedure statements.
Procedure statement	Each procedure statement begins on a new line, but can be continued onto multiple lines. It consists of an optional label followed by a command.
Label	A label identifies a specific statement. The label is optional. If used, place the label before a procedure statement and follow the label immediately (without intervening spaces) by a colon (:).
Command	A command identifies the kind of procedure statement. You can follow a command by keywords, functions, constants, names, expressions, and comments.
Function	A function is a subprocedure that derives a value or condition from other data.
Keyword	A keyword identifies how certain parameters are used. Most keywords are optional.
Keyword operand	A keyword operand (or operand) identifies the data values associated with a keyword. In the procedure statement, follow a keyword with one or more spaces and a keyword operand. For example: *keyword1 operand keyword2*
Keyword phrase	A keyword phrase is the name given to the combination of a keyword plus its operand.

Spaces

Use one or more spaces to separate the command, keywords, and operands, unless otherwise stated (syntax rules) or unless a special character (such as an arithmetic operator) is present.

The following is an example of an ASL command using operators as separators between the operands on the IF command.

```
IF A+B=C+D THEN
```

To make the procedure statement easier to read, use blanks between the operands and operators. For example:

```
IF A + B = C + D THEN
```

Continuation

You can write procedure statements on multiple lines. Terminate each line (ignoring comments) by a blank space followed by a comma. Continue the remainder of the statement on the following lines. This is useful for clarity. For example:

```
IF      NAME    =       'THE ABC COMPANY'  ,
 AND    NUMBER  =       '00001' ,
 OR     NUMBER  =       '00002' ,
 OR     NUMBER  =       '00003' ,
THEN
```

Constants

There are six types of constants: character, integer, decimal, floating point, time, and pattern. Each type of constant is described in the following pages.

Character Constants

Delimit character constants with single quotation marks. For example:

The Value	The Character Constant Representation
A	'A'
	' '
123	'123'
THE X	'THE X'
CAN'T	'CAN''T'

Specify a single quotation mark within a constant with two consecutive single quotation marks.

Integer Constants

Specify integer constants with only numeric digits only.

- If the integer is negative, place a minus (-) sign before the integer constant.
- If the integer is positive, place an optional plus sign (+) before the integer constant. For example:

```
+100
  -2
   0
```

Decimal Constants

Specify decimal constants with numeric digits, an optional sign, and a decimal point.
If you use a plus (+) sign or a negative (-) sign, make it the first character in the constant. For example:

```
-200.0
   3.99
   -.05
+123.456
```

Floating Point Constants

Specify floating point constants with an optional sign preceding an integer or decimal constant followed by a power of ten expressed in exponential notation. For example:

```
 1.50E10
13.75E-5
-5200E+5
```

Time Constants

Specify a time constant, HH:MM:SS.n...n (hours, minutes, seconds, decimal seconds), by the letter T, followed by a beginning single quotation mark, the time constant, and a single closing quotation mark. For example:

```
T'23:16:11'
T'12:01:00.125'
T':09:10.5'
```

Patterns

Specify a pattern starting with the letter P, followed by a beginning single quotation mark, the string of special symbols, and ending with the closing single quotation mark. For example:

```
P'ZZZ999'
P'#(#999#)#999#-#9999'
```

For more information on pattern symbols, see Appendix B, "Technical Notes".

Names

Field names must follow these rules:

- Begin field names and statement labels with an alphabetic character (A-Z).
- If the name is more than one character, make the remaining characters either alphabetic characters (A-Z), numeric digits (0-9), or underscores (_).
- Enclose field names that do not conform to this syntax in double quotation marks.

The following are valid field names or procedure statement labels:

```
CUST_NUM
BAL30
CUSTNO
```

Qualifiers

You can prefix a name with a standard 1-character qualifier and a period. A qualifier identifies a specific file or usage of a field. Qualifiers relate fields to their origins:

Qualifier	Type of Location	
Blank or N	New master file.	
1-9	Coordinated files 1-9.	
T	Temporary file.	
F	Flag field.	
X	Transaction file	For VISION:Builder and VISION:Two.
O or 0	Old master file.	
W	Working storage	
V	Linkage section.	
A, B, E, H, J K, M, Q, 1-9	Array (must match the qualifier that identifies the array.	

The following are valid qualified field names:

```
T.TEMP
N.CUST_NUM
1.CORD_FLD
F.DATE
```

Example 1

```
T.TEMP
```

This is a reference to a temporary field named TEMP.

Example 2

```
1.CORDFLD
```

This is a reference to a field named CORDFLD from the file in the application that has been assigned the qualifier 1.

See Appendix B, "Technical Notes" for a list of all valid qualifiers and their origins.

Names with Special Characters

Enclose names that include special characters in double quotation marks. The following is an example of a field name with a special character (an embedded blank):

```
N."DATE ONE"
```

Without the special notation, the blank in the field name is interpreted as two separate fields instead of one.

Comments

Place comments anywhere following a semicolon (;), except on a continued line. For example:

```
;       RESETTING A TEMPORARY FIELD
;
LET T.REGION = '       '  ;  RESET FIELD TO BLANKS
```

The first two lines show comments on lines by themselves. The third line is an example of a comment on a command line. Separate the comment from the command and its keyword phrase by a semicolon.

Arithmetic Expressions

Code arithmetic expressions using the operators +, –, *, and / for addition, subtraction, multiplication, and division, respectively. For example:

```
A+B    A-B      B*C    D/C
```

As in conventional algebraic notation, operations within an arithmetic expression are executed from left to right. However, multiplication and division are performed prior to addition and subtraction unless this order is overridden by parentheses. For example:

```
A-B*C+D
```

In this arithmetic expression, the multiplication between field B and field C is performed before the addition and the subtraction.

When you write the expression as `(A-B)*(C+D)`, the addition and subtraction are performed before the multiplication.

To evaluate the difference, assume that:

```
A=5
B=3
C=2
D=1
```

for the two arithmetic expressions given in the prior example. Figure 2-1 shows the steps taken to come up with the result of the arithmetic expressions.

Without Parentheses	With Parentheses
A-B*C+D	(A-B)*(C+D)
5-3*2+1	(5-3)*(2+1)
5-6+1	2 * 3
0	6

Figure 2-1 Arithmetic Expressions

Logical Expressions

Use logical expressions in IF, CASE, and DO commands.

Make a logical expression from conditional functions, relational expressions, or list expressions connected by logical operators.

Conditional Functions

A conditional function is a function that returns a true or false condition. For example:

```
VALIDATE(FIELD ORDRDATE DATE)
```

shows the VALIDATE function. The date validation of the field ORDRDATE will either be true or false depending on the contents of ORDRDATE being a valid date.

In the following examples, A can be a field name, and B and C can be any constant, field name, or arithmetic expression.

Relational Expressions

A relational expression is composed of two values connected by a logical operator. In relational expressions, you can represent logical operators as characters or as symbols:

EQ or =
NE or <>
GT or >
LT or <
GE or >=
LE or <=

Examples

```
A  EQ  B  (equal)
A  NE  B  (not equal)
A  LT  B  (less than)
A  GT  B  (greater than)
A  LE  B  (less than or equal to)
A  GE  B  (greater than or equal to)
```

List Expression

A list expression lists the specific values to test. For example, `NUMBER EQ 00001 00002 00003 00004` specifies that NUMBER should be compared against the four values listed.

In a list expression, use only the EQ and NE operators. In relational or list expressions, you can represent logical operators operators as characters or as symbols:

EQ or =
NE or <>
GT or >
LT or <
GE or >=
LE or <=

Boolean Logical Operators

There are three Boolean logical operators: AND, OR, and NOT. Use parentheses to specify the order of evaluation.

Example 1 The OR operator.

```
A=B OR A=C
```

If either relational expression is true, the logical expression is true.

Example 2 The AND operator.

```
A=B AND A=C
```

Both relational expressions must be true for the logical expression to be true.

Example 3 The NOT operator.

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

Both relational expressions must be true for the logical expression to fail. The whole expression is true if the expression in parentheses is false. In other words, the NOT operator reverses the evaluation of true or false for the final outcome of the expression.

The NOT operator is best understood when used in a conditional function such as VALIDATE. For example, `NOT VALIDATE(DUEDATE DATE)` is true when the field DUEDATE is not valid.

You can also use the NOT operator to exclude a few values that are not wanted for processing instead of including all of the values that are wanted. For example, the following is true for all values of NUMBER except 1 or 5:

```
NOT (NUMBER = 1 OR NUMBER = 5)
```

The following table shows numeric examples and the subsequent true or false evaluations of logical expressions.

	A=B	A=C	A=B OR A=C
A=1, B=2, C=2	FALSE	FALSE	FALSE
A=2, B=2, C=1	TRUE	FALSE	TRUE
A=2, B=2, C=2	TRUE	TRUE	TRUE

	A=B	A=C	A=B AND A=C	NOT (A=B AND A=C)
A=1, B=2, C=2	FALSE	FALSE	FALSE	TRUE
A=2, B=2, C=1	TRUE	FALSE	FALSE	TRUE
A=2, B=2, C=2	TRUE	TRUE	TRUE	FALSE

Statement Syntax

This section describes the conventions used to provide a precise description of the syntax of a function or command.

Enter commands and functions in the exact order given on the syntax line.

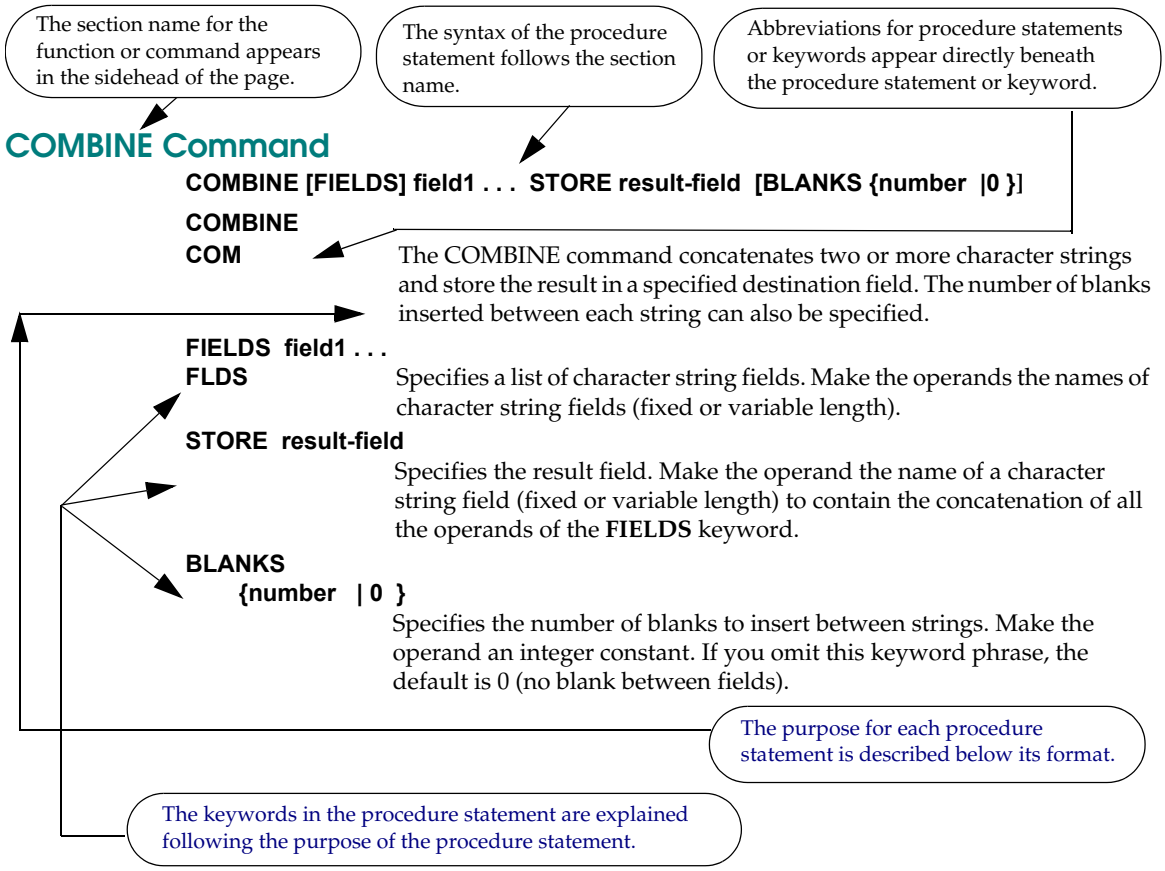
The following syntax conventions apply to ASL statement syntax. As the examples given with each command show, you enter your procedures using uppercase letters.

- Brackets [ ] indicate an optional parameter.
- Braces { } indicate a choice of entry; unless a default is indicated, you must choose one of the entries.
- Required parameters do not have brackets or braces surrounding them.
- Items separated by a vertical bar (|) represent alternative items. Select only one of the items.
- An ellipsis (. . .) indicates that you can use a progressive sequence of the type immediately preceding the ellipsis. For example: name1, name2, ...
- Bold, uppercase type indicates the characters to be entered. Enter such items exactly as illustrated. You can use an abbreviation if it is indicated.
- Italic, lowercase type specifies fields to be supplied by the user.
- Underscored type indicates a default option. If you omit the parameter, the underscored value is assumed.
- Separate commands, keywords, and keyword phrases by blanks.
- Enter punctuation such as parentheses and colons exactly as shown.

Note: The section heading for each function and command appear at the top of a page.

Page Layout

The following describes the format used to present each function or command.



Example:

```
COMBINE FIRSTNAM LASTNAME  STORE NAME  BLANKS 1
```

The Nature of ASL

ASL enables you, the application developer, to work with a logically related set of data. The objective is for the application developer to focus upon the algorithms and business considerations of the problem independently of the way the data is physically structured or organized. Thus, for the most part, the application developer can write ASL statements without concern for the structure of a record and without concern as to how the data is accessed.

Underlying ASL, therefore, are several important considerations that enable the application developer to process the data correctly and consistently.

In developing an application in ASL, the application developer merely references fields by name. The ASL processor locates all fields that are applicable. More importantly, the ASL processor ensures that when data is requested from several different fields, and even from different databases, only the correctly related data is made available. In this manner, only correct and consistent sets of data are processed.

The application developer should be aware of the following important and powerful capabilities built into the ASL processor and the ASL language:

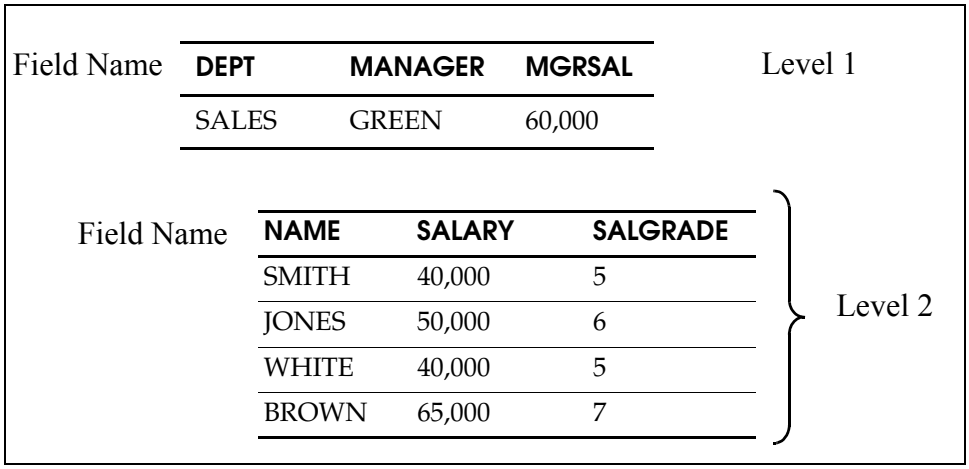
- Implicit loops or set operation
- Record and segment processing

Implicit Loops and Set Operation

ASL is a set language. This means that one statement of ASL processes all occurrences of the data (that is, the complete set of occurrences). In a structured record, repeated segments of data represent different occurrences of the data. In most business situations, the application developer wants to apply a business process (or algorithm) to all occurrences of the data, or at least to specifically selected occurrences.

The ASL processor actually applies the ASL statements to each set of data in turn. The application developer can write the algorithm as if it applied to only one instance of the data. In practice, all instances are selected.

Consider the simple structure shown in Figure 2-2.



Field Name

DEPT	MANAGER	MGRSAL
SALES	GREEN	60,000

Level 1

Field Name

NAME	SALARY	SALGRADE
SMITH	40,000	5
JONES	50,000	6
WHITE	40,000	5
BROWN	65,000	7

Level 2

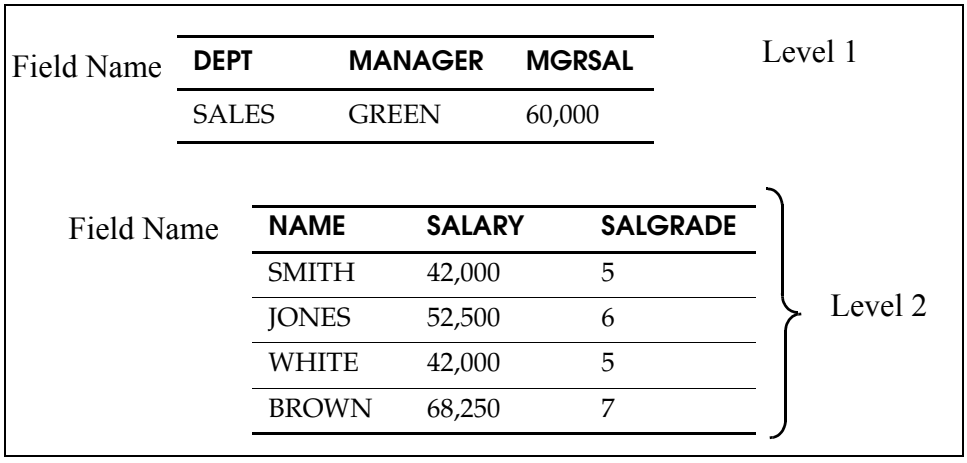
Figure 2-2 Employee File

This structure is typical of a department level with subordinate segments, one for each employee. To increase the salary of everybody in the department by 5 percent, use the following ASL statement:

```
LET SALARY = 1.05*SALARY
```

While this looks like one statement, it carries with it the implied "for all occurrences of SALARY."

The result of this one statement is shown in Figure 2-3.



Field Name

DEPT	MANAGER	MGRSAL
SALES	GREEN	60,000

Level 1

Field Name

NAME	SALARY	SALGRADE
SMITH	42,000	5
JONES	52,500	6
WHITE	42,000	5
BROWN	68,250	7

Level 2

Figure 2-3 Update All Records in the Employee File

The single statement acted on all occurrences of the data.

If you have your original data of Figure 2-2, but you need to increase the salaries of those people whose salary grade is less than 7, use the ASL statements shown below.

```
IF SALGRADE LT 7
   LET SALARY = 1.05*SALARY
END
```

The result of this statement is shown in Figure 2-4

.

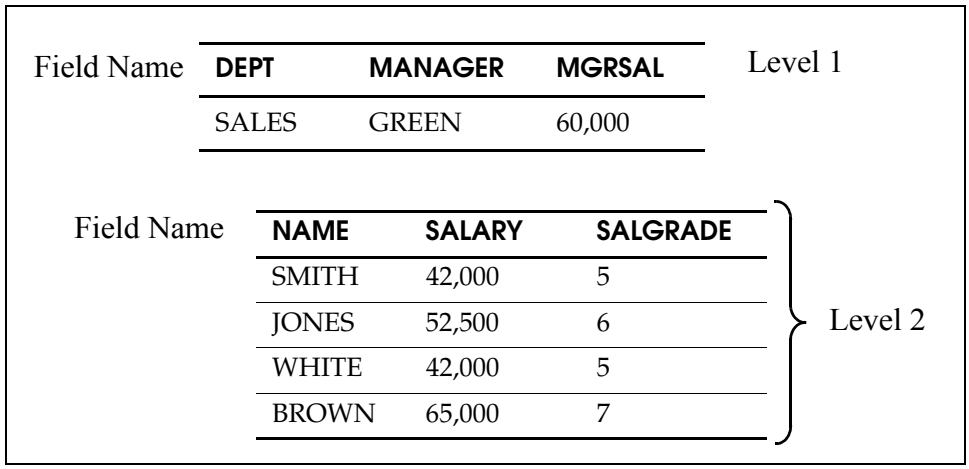


Figure 2-4 Update Selected Records in the Employee File

The ASL processor identifies all segments of data that are relevant and executes the statement. As the application developer, you do not have to write any complex looping structures.

As another example, assume that the three level record structure shown in Figure 2-5 exists.

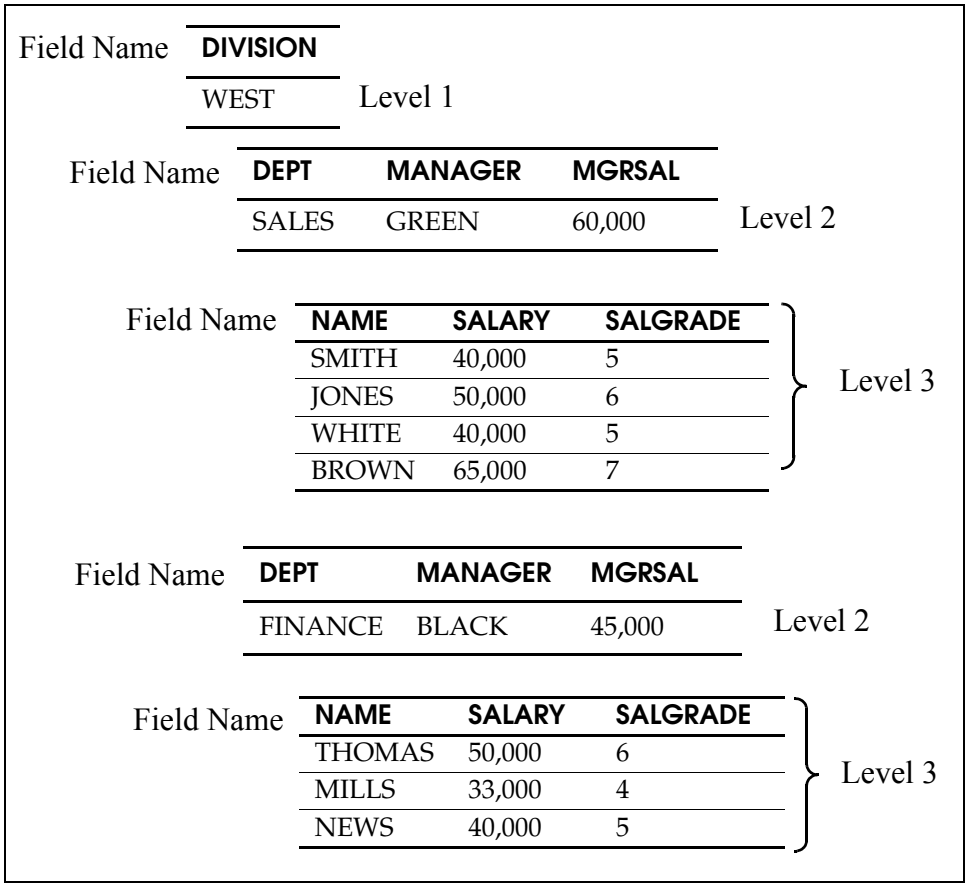


Figure 2-5 Three Level Record Structure

To select every employee who earns more than their manager, use the following ASL statements:

```
IF SALARY GT MGRSAL
   CALL REPORT HIGHPAID
END
```

These statements select the following sets of data for the report HIGHPAID:

BROWN	65,000	7

THOMAS	50,000	6

In processing these ASL statements, the ASL processor maintains the integrity of the relationships of the data. Each employee is associated with the appropriate manager. No employee is associated with somebody else's manager. The data that is returned and processed is that which you logically would expect.

ASL, therefore, is a set language that processes all occurrences of the data that satisfy any existing criteria. ASL does not stop with the first occurrence of the data. Any procedural statements are processed against each occurrence of the data that satisfies the selection criteria. When this is done, ASL searches for the next occurrence of the data that satisfies the selection criteria. This process occurs until all instances of the data have been processed. Some application developers may like to think of this as the creation of an implicit loop based upon the number of segment occurrences that exist.

In the vast majority of cases, application developers can write an algorithm or business consideration as if there were only one occurrence of the data and be secure in the knowledge that all appropriate occurrences will be processed.

The examples show that one or two statements can process large amounts of data. There are no complicated data navigation commands; it was all transparent. You can focus on the problem and leave the data to ASL.

In some instances, you must know that you are, in fact, dealing with a set of data and not an individual occurrence. Under some circumstances, the following specific situations can occur:

- Stop the procedural algorithm working on the particular occurrence of the data and move on to the next occurrence.
- Stop all processing of any future occurrences of the data.

Sometimes these situations are relevant to the particular problem being solved, while in other cases there are important performance considerations.

When processing large structured records, for example, you can improve performance by bypassing processing of segments where it is known that no action is required.

Controlling the Processing of Repeated Segments

ASL provides commands for the application developer to control the processing of repeated segments within a structure. Two particular commands are of importance: CONTINUE and LEAVE.

- The CONTINUE command enables the application developer to stop the algorithm working on the current occurrence of the data and move to the next occurrence.
- The LEAVE command causes the procedure to stop processing this occurrence of the data and to bypass all other occurrences of the data within the current set for the current procedure.

Within ASL, any selection statement (such as IF, DO) limits the available segments of data for processing by subsequent procedural statements. This applies also to subroutine procedures, reports, or subfiles. Any object that is called from another object inherits the same view of the data as currently defined by the parent object. All segments beyond the scope of this view are still available for processing. Once again, the consistency of data is maintained.

Record and Segment Processing

The set theory philosophy, as described above, says that the application developer should not worry about how many occurrences of the data exist nor the relationships between the various pieces of data. That is the responsibility of ASL and the ASL processor. The ASL processor guarantees that all occurrences of the data are available for processing with the correct relationships between the data.

This philosophy continues in ASL with the philosophy of handling records and segments within a record. The ASL user does not have to be concerned with where the data physically exists. ASL is structured to be able to handle records and segments in a similar manner.

Records are normally thought of as separate physical entities; whereas, segments are thought of as being contained within a record and being repetitions of data.

The Distinction Between Records and Segments

The distinction between records and segments is an artificial one imposed upon the application developer by virtue of hardware constraints. Many records in a file really constitute repeats of root segments. In effect, they are no different from repeats of subordinate segments. Unfortunately, because of the difference in nature of the physical arrangement of the data, it frequently calls for different programming techniques to access the data. ASL and the ASL processor remove these restraints from the application developer.

For the most part, you, the application developer, can write your application as if you were dealing with an infinite space of data. All data is treated as if the fields belong to a segment. It is the responsibility of ASL and the ASL processor to work

out whether accessing a segment is merely an act of navigating through a structured record or whether a physical read of a new record from the file must be made. All of this is transparent to the application developer.

Automatic Navigation

Navigation through records and segments is automatic for a majority of applications. As described above, a reference to a field causes navigation through a record for all suitable occurrences of segments containing that data. When all segments have been processed, the ASL processor automatically reads the next record and transfers control to the beginning of the processing cycle.

Application-Controlled Navigation

Under certain circumstances, an application developer might not want to have the ASL processor perform the automatic navigation through the data. If this is the case, the application developer can use explicit functions (for example, FIND) to locate specific instances of segments.

When you issue a FIND command for a segment, the ASL processor locks onto the segment that is identified or located. All subsequent procedural statements can only access data within that segment. If you require access to all the occurrences of the segment after a FIND command completes, you must issue a RELEASE command.

ASL provides powerful commands for those applications where the application developer needs to control the navigation through the data.

Defining ASL Procedures

You can start ASL through VISION:Workbench for DOS or VISION:Workbench for ISPF (OS/390 (z/OS) only). You can use ASL instream in place of, or in addition to, traditional VISION:Inform requests.

Using ASL in VISION:Workbench for DOS

When you use VISION:Workbench for DOS, enter ASL procedures on the Procedure Definition window. A sample screen is shown in Figure 2-6.

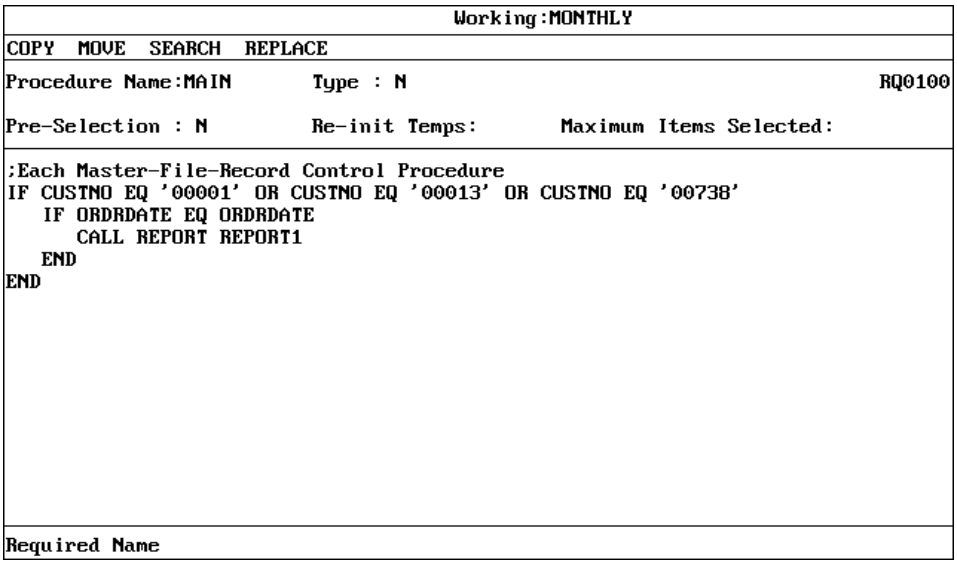


Figure 2-6 ASL in VISION:Workbench for DOS

Using ASL in VISION:Workbench for ISPF

You can also enter ASL procedures in the VISION:Workbench for ISPF Procedure Definition window. A sample screen is shown in Figure 2-7.

```
ASL4 ---- MISPF.SOURCE(MONTHLY)          ---------------------------- ROW 1 OF 12
COMMAND ===>                                                  SCROLL ===> DATA
                                                           PROCEDURE: MONTHLY

          Procedure Type ===> _          Max Items ===> ____
          Reinit Temps?  ===> _

Line  Free Form Processing Statements
Cmd   --------------------------------------------------------------------------
''''  IF QANONHND < 5
''''     CALL SUBFILE SUBFILE1
''''  END
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
''''  ____________________________________________________________________________
********************************* BOTTOM OF DATA ********************************
```

Figure 2-7 ASL in VISION:Workbench for ISPF

Using ASL Instream

Although ASL is generally used through VISION:Workbench for DOS or VISION:Workbench for ISPF, you can also enter ASL statements instream.

1. Place ASL statements after an ER statement.
2. On the next line, place two system delimiters followed by PROC, starting in column one.
3. Enter the ASL procedure.
4. Place two system delimiters followed by PEND, starting in column one on the line after all of the ASL statements for the procedure.

Figure 2-8 is an example of instream ASL statements.

```
procname ER
##PROC
STOCKVAL: FIELD P 5 DEC 2
IF QANONHND > 0
  LET T.STOCKVAL = QANONHND * UNITCOST
  CALL REPORT VALUEREP
END
##PEND
```

Figure 2-8 Instream ASL Statements

You can use ASL procedures and traditional requests (PR statements) in the same application. However, you cannot use PR statements and ASL in the same procedure or request.

Chapter

3

Built-in Functions

A function is a subprocedure or subprocess that derives a value or condition from other data. The function value is created when reference is made to it. You use the function as if it is a field. You write a function differently than a field, but basically you use a function wherever you use a field name.

Value Functions and Conditional Functions

Functions return either "value" (that is, they represent a field value) or "conditional" (that is, they represent a true or false condition). When needed, you use functions in procedure statements.

- You can use value functions anywhere a field name can be used.
- You can use conditional functions anywhere a conditional operand is needed.

Specifying Functions

There are five built-in functions available in ASL.

You specify functions by entering a function name **followed immediately (with no intervening spaces) by a left parenthesis**, one or more keyword phrases, and terminating with a right parenthesis.

Longer function and keyword names have abbreviations that are defined directly beneath the function or keyword in the following descriptions.

Types of Built-in Functions

There are two types of built-in functions: conditional and value.

Conditional Functions

All conditional functions are evaluated and return a true or false condition, but some functions also perform other actions. The conditional functions are as follows:

FIND Find a segment function.

This function returns a true condition when a segment or record exists. If the segment or record does not exist, the function returns a false condition.

If the segment or record exists, the function locates the segment or reads the record so that subsequent field values can be obtained from the segment.

LOCATE Locate a cell, row, or column in an array.

This function returns a true condition when the row and/or column specifications are within the bounds of the array. Otherwise, the function returns a false condition.

If the function returns a true condition, the specified cell, row, or column is available for subsequent processing.

Note: LOCATE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

SCAN String scan function.

This function searches a field for a character or set of characters.

- If the characters being searched for are found, the function returns a true condition.
- If the characters are not found, the function returns a false condition.

The location where the characters are found is kept in flag fields when the function is true. When the function is false, the flag fields are not set.

VALIDATE
VAL

Field validation function.

This function validates the contents of a field for either a valid calendar date or for a predefined pattern of characters.

- If the date is valid or the pattern is found, the function returns a true condition.
- If the content is not a valid date or the pattern is not satisfied, the function returns a false condition.

Value Functions

Value functions return an actual value, either a result value from a table or a part of an existing field. The value functions are as follows:

LOOKUP
LU

Table lookup function.

This function processes external tables by searching an argument list and returning back a result value.

PF

Partial field function.

This function isolates part of a character field.

The Built-in Functions

This section describes each function.

After the syntax and operands of a function are explained, several examples of the function are given. Values for the fields are given, as well as other pertinent information and the result of the procedure statements.

When field values are shown, they are enclosed in single quotation marks. For example, the contents of the fields CHARS, T.NUMBER, LS, and LN are enclosed in single quotation marks, but the quotation marks are not part of the field.

```
CHARS       =     'ABCDEFGHIJKL'
T.NUMBER    =     '3'
LS          =     '2'
LN          =     '10'
```

The quotation marks delimit the beginning and end of the field.

For the rules and explanations of the statement syntax and page layout, see Chapter 2, "Terminology, Syntax, and Processing".

FIND Function (Conditional)

FIND([SEGMENT]*segment-name*
[FIRST | LAST | NEXT | WHERE *selection-expression*])

Use FIND as a conditional function in a logical expression to establish the existence of an occurrence of a segment or record.

- If an occurrence of the segment or a record exists, the FIND function returns a true condition and that segment or record is used for processing in the current procedure.
- If an occurrence of the segment or a record cannot be found, the FIND returns a false condition and reference should not be made to fields on the missing segment or record.

There are five keywords:

SEGMENT *segment-name*
SEG

Specifies the name of the segment to locate.

- If the segment-name is a root segment, a record is read from the file.
- If the segment-name is a lower level segment, a particular occurrence of the segment is located.

If the WHERE phrase is not present, the following rules apply:

- The first FIND function for a segment in the procedure locates the first occurrence of the segment.
- Subsequent FIND functions for the same segment in the same procedure locate segments in the order that they are in the record. A subsequent FIND function for a record reads the next record on the file. When the subsequent FIND function is processed, the segment previously found is no longer available. In other words, only one occurrence is processed at any given time.
- For some databases (for example, DB2®), the concept of "first" and "order" might have no meaning. In these cases, the FIND function will locate an occurrence of a segment/record. Subsequent FIND functions will locate a different occurrence of the segment/record.

FIRST

Causes the first occurrence of a lower level segment to be located for processing. This keyword is not permitted for memory optimized (MOSAIC) processing of relational or Generalized Data Base Interface (GDBI) files.

LAST

Causes the last occurrence of a lower level segment to be located for processing. This keyword is not permitted for memory optimized (MOSAIC) processing of relational or GDBI files.

NEXT

Causes the next available occurrence of a lower level segment to be located for processing. The first FIND NEXT function of a procedure locates the first occurrence of a segment.

WHERE *selection-name*

Specifies the value of the segment or record key to locate. Use a logical expression as the WHERE operand. Use a primary key field name for the segment or record as one of the field names in the logical expression.

For a lower level segment, the WHERE phrase causes only one segment to be located: the segment whose primary key is equal to the value specified in the operand.

For a root segment, the WHERE phrase causes only one record to be read and is only allowed for a direct-read file. You can use two types of reads: the key of the record equal to the value specified in the operand, or the key of the record whose value is greater than or equal to the value specified in the operand.

The FIND function is a conditional function you can use anywhere a logical expression. The examples show the FIND function on an IF command. For more information on the IF command, see Chapter 4, "Procedure Statements".

Example 1

```
IF FIND(1.SEGX)
  LET T.TEMP1 = 1.FIELD1
ELSE
  LET T.TEMP1 = 0
END
```

The FIND function reads a record from file 1. If such an occurrence is found, the FIND function is true and the next statement is executed; if no occurrence is found, the FIND function is false and the ELSE clause is executed.

Example 2

```
IF FIND(SEGMENT ORDER WHERE ORDERNUM = '12345')
   CALL PROC CHKORD
END
```

The FIND function searches for an ORDER segment with the key value of 12345. ORDER is a lower level segment in the file.

- If the function finds a segment, the PROC CHKORD is called.
- If the function does not locate the segment, processing continues with the statement after the END command.

LOCATE Function (Conditional)

Note: LOCATE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

LOCATE([ARRAY] *array-identifier* {[Row row-number] [COLUMN column-number]})

Use LOCATE as a conditional function in a logical expression to locate a cell, row, or column in an array. It operates exactly like the LOCATE command, except that the function returns a true or false. You use this condition in evaluating the logical expression.

Use the function in place of the command where the row and/or column parameters might cause the LOCATE command to fail. Use the LOCATE function to perform an array location operation and test for its success or failure in one operation.

For a discussion of the LOCATE function keywords, see LOCATE Command.

Example

```
IF LOCATE(ARRAY A ROW T.Y  COLUMN T.X)
  ; process cell
ELSE
  ; Handle error situation - check ASTATUS flag field
END
```

SCAN Function (Conditional)

**SCAN([FIELD] *field-name* [FOR] *search-value/pattern*
{FROM LEFT | FROM RIGHT} [NOTEQUAL])**

SCAN is a conditional function. This function checks a field to see if a specified field contains a specific character string or pattern of characters. If a match is found, a true condition for the function is established. If no match is found, the condition is false.

When the function finds the character string or pattern, the location of the search-value/pattern within the field being scanned is stored in the following flag fields:

Field Flag Name	Long Form Field Flag Name	Description
LS	LSTART	Left part Start
LN	LNUMBER	Left part Number of characters
MS	MSTART	Middle part Start
MN	MNUMBER	Middle part Number of characters
RS	RSTART	Right part Start
RN	RNUMBER	Right part Number of characters

The scanned field can be considered as having the following parts:

- Search-value/pattern.
- Part of the field to the left of the search-value/pattern.
- Part of the field to the right of the search-value/pattern.

The flag fields identify the starting location and length of these three parts of the field being scanned.

- LS contains the starting location of the part of the field being scanned that is to the left of the search-value/pattern.
- LN contains the length of this left portion of the field.
- MS contains the location of the beginning of the search-value/pattern within the field being scanned.
- MN contains the length of the search-value/pattern.
- RS contains the starting location of the part of the field being scanned that is to the right of the search-value/pattern.
- RN contains the length of this right portion of the field.

There are four keywords:

FIELD *field-name*
FLD

Specifies the field to scan. Specify a name or PF function as the operand.

FOR *search-value/pattern*

Specifies the value or pattern for which to scan. Specify a name, character constant, or validation pattern as the operand. For valid pattern symbols, see Appendix B, "Technical Notes".

FROM LEFT | FROM RIGHT

Specifies the direction of the scan. If you omit this keyword phrase, LEFT is assumed.

NOTEQUAL
NE

If you add the NOTEQUAL keyword to the SCAN function, the function returns a true condition when the search-value/pattern is not found.

If you do not enter the NOTEQUAL keyword, the SCAN function returns a true condition when the search-value/pattern is found. This keyword has no operands.

Example 1

Assume these values for the following fields:

	ALPHA	=	'ABCCCAB123ABEEE'			
Initial Values:	LS	=	'0'	LN	=	'0'
	MS	=	'0'	MN	=	'0'
	RS	=	'0	RN	=	'0'

		RESULTS					
Function	**Condition**	**LS**	**LN**	**MS**	**MN**	**RS**	**RN**
SCAN(ALPHA FOR 'AB' FROM LEFT)	TRUE	01	00	01	02	03	13
SCAN(ALPHA FOR 'AB' FROM RIGHT)	TRUE	01	10	11	02	13	03
SCAN(ALPHA FOR 'DE' FROM LEFT)	FALSE	01	15	16	00	16	00
SCAN(ALPHA FOR P'999' FROM LEFT)	TRUE	01	07	08	03	11	05
SCAN(ALPHA FOR P'999' NE)	FALSE	00	15	16	00	00	00

The first occurrence of the search value/pattern located in the field causes the SCAN function to be true, and the flag fields are set to that location.

The next two examples of the SCAN function show it being used in the IF command where a logical expression is necessary. You can use the SCAN function in a logical expression, because it sets a true or false condition. For more information on the IF command, see Chapter 4, "Procedure Statements".

Example 2

```
IF SCAN(T.MSG FOR 'XYZ')
   LET T.LEFTPART = PF(T.MSG LS LN)
END
```

If the field T.MSG contains THE XYZ COMPANY, the SCAN function is evaluated as true, because the letters XYZ were found in T.MSG. As a result of the expression, T.LEFTPART contains the value THE b/. Because the FROM phrase is omitted, FROM LEFT is assumed.

Example 3

```
IF SCAN(CUSTNO FOR P'99999')
   CALL REPORT VALID
ELSE
   CALL REPORT INVALID
END
```

The field CUSTNO is a 5-byte character field. The pattern P'99999' checks for numeric characters. If the field CUSTNO contains any non-numeric characters, the SCAN function is evaluated as false.

VALIDATE Function (Conditional)

VALIDATE([FIELD] *field-name* {PATTERN P'*pattern*' | DATE})

VALIDATE
VAL

Specify conditional function. It validates a field for a pattern or date.

- If the field contains a valid date or its content matches the specified validation pattern, the function is true.
- If the field does not contain a valid date or the specified validation pattern, the function is false.

Use the VALIDATE function as a logical expression in procedure statements. There are three keywords:

FIELD *field-name*
FLD

Specifies the field to be validated. Use the name of a field or a PF function of a field as the operand.

PATTERN P'*pattern*'
PAT

Specifies a pattern to use for the validation.

- Specify a pattern starting with the letter P, followed by a beginning single quotation mark, a string of special pattern symbols, and a closing single quotation mark.
- Use up to 30 characters in the operand.
- Code one pattern symbol for each character of the field being validated.

For valid pattern symbols, see Appendix B, "Technical Notes".

The function checks each character in the specified field against its corresponding pattern character. This keyword is mutually exclusive with the DATE keyword.

DATE

Checks that the field contains a valid calendar date. This keyword causes the function to validate the content of the field for a valid calendar date.

- If the field contains a valid date, the function is evaluated as true.
- If the field contains a non-calendar date or the date is not in the format expected by the function (as defined to the system), the function is evaluated as false.

The type of date storage (for example, Julian and MMDDYY) is defined in M4SFPARM. This keyword has no operands and is mutually exclusive with the PATTERN keyword.

Example 1

Function	If ORDRDATE contains...	Condition
VALIDATE(ORDRDATE PAT P'999999')	'123456'	TRUE
VALIDATE(ORDRDATE DATE)	'987654'	FALSE
VALIDATE(ORDRDATE PAT P'999999')	'011595'	TRUE
VALIDATE(ORDRDATE DATE)	'011595'	TRUE
VALIDATE(ORDRDATE PAT P'999999')	'ABCDEF'	FALSE
VALIDATE(ORDRDATE DATE)	'ABCDEF'	FALSE

The next two examples show the conditional VALIDATE function in the IF statement, which requires a logical expression. See Chapter 4, "Procedure Statements" for more information on the IF command.

Example 2

```
IF VALIDATE(FIELD ORDRDATE DATE)
   CALL REPORT ORDER
ELSE
   LET T.MSG ='ILLEGAL DATE'
END
```

If the field ORDRDATE contains 999999, the VALIDATE function would be evaluated as false, because 999999 does not represent a valid calendar date.

Example 3

```
IF VALIDATE(PF(NUM  5  4)  P'9999')
   LET T.COUNT = PF(NUM 5 4)
END
```

If the field PF(NUM, 5, 4) contains 5678, the VALIDATE function is evaluated as true, because NUM contains numerics starting in position 5 for a length of 4. As a result, T.COUNT is set to 5678.

LOOKUP Function (Value)

LOOKUP([TABLE] *table-name* [ARGUMENT] *lookup-argument*
[NEAREST | SMALLER | LARGER | INTERPOLATE])

Note: If you do not chose any of the following keywords, the arguments in the table are searched for a value equal to the value in lookup-argument.

LOOKUP
LU

Is a value function that takes the given argument value, searches the argument list in the table to locate a specific argument, and returns the corresponding result value.

Because LOOKUP is a value function, you can use it anywhere you use a field name or constant. There are six keywords.

TABLE *table-name*
TAB

Specifies the name of a table. Specify the operand to be the name of a table definition defined to the VISION:Inform library.

ARGUMENT *lookup-argument*
ARG

Specifies the field containing the argument value. Use a field name as the operand.

NEAREST
NRST

Causes the arguments in the binary table to be searched for a value equal to or nearest the value in lookup-argument. This keyword has no operands.

SMALLER
SMLR

Causes the arguments in the binary table to be searched for a value equal to or smaller than the value in lookup-argument. This keyword has no operands.

LARGER
LRGE

Causes the arguments in the binary table to be searched for a value equal to or larger than the value in lookup-argument. This keyword has no operands.

INTERPOLATE
INT

Performs linear interpolation on the argument list in the binary search table to determine the argument/result to be selected. This keyword has no operands.

As the LOOKUP function represents a value, you can use the function anywhere you would use a field name or a constant. In the examples, the LOOKUP function is used on a LET command. For more information on the LET command, see Chapter 4, "Procedure Statements".

Example 1

```
Table Name:  MONTHS
```

Argument list	Result list
01	JANUARY
02	FEBRUARY
03	MARCH
04	APRIL
05	MAY
06	JUNE
07	JULY
08	AUGUST
09	SEPTEMBER
10	OCTOBER
11	NOVEMBER
12	DECEMBER

MM is a field in a file whose contents is 11.

```
LET T.MONTH = LOOKUP(MONTHS, ARGUMENT MM)
```

After the LOOKUP function, T.MONTH contains NOVEMBER.

Example 2

```
Table Name: GRADES
```

Argument list	Result list
1.0	FAILING
1.5	BELOW AVERAGE, NEEDS IMPROVEMENT
2.0	BELOW AVERAGE
2.5	AVERAGE
3.0	GOOD
3.5	EXCELLENT
4.0	PERFECT

SCORE is a field in the file whose content is 3.8.

```
LET T.EVALUATE = LOOKUP(GRADES ARGUMENT SCORE NRST)
```

After the LOOKUP function, T.EVALUATE contains PERFECT.

PF Function (Value)

PF([FIELD] *field-name* [START] *start-position* [LENGTH *partial-length*])

PF

Is a value function that defines the value as a part of the character field against which the function is used. You can use the PF function on a character field and in any replacement or logical expression. There are three keywords:

FIELD *field-name*
FLD

Specifies the name of a character field (fixed or variable length). The PF function returns a value that is only part of the field specified in this operand.

START *start-position*

Is a required entry that specifies the starting character position of the value within the previously named field, where byte 1 is the beginning of the field. This operand can be an integer or one of the flag field names LS, LN, MS, MN, RS, or RN (see SCAN Function (Conditional).

LENGTH *partial-length*
LEN

Is required for a fixed length field, but is an optional entry for a variable length field. The result of the PF function is a partial value from the field named in the first operand. The partial-length specifies how many characters the partial value is to contain. This operand can be an integer or one of the flag field names LS, LN, MS, MN, RS, or RN (see SCAN Function (Conditional).

You can omit the keyword LENGTH and its operand for variable length fields, in which case, the remaining length of the field from the start position is assumed.

Example 1

Assume these values for the following fields:

```
CHARS      =     'ABCDEFGHIJKL'
T.NUMBER   =     '3'
LS         =     '2'
LN         =     '10'
```

Function	Partial Field Value
PF(CHARS START 1 LENGTH 7)	'ABCDEFG'
PF(CHARS 3 6)	'CDEFGH'
PF(CHARS LS 4)	'BCDE'

The PF function on the CHARS field causes the value of the field to be a subset of the original contents of the field.

Example 2

Field: CUSTNO PF(CUSTNO START 1 LENGTH 1)

Field contents: '1000' '1'

```
IF PF(CUSTNO START 1 LENGTH 1) = '1' THEN
   CALL REPORT CUSTOMER
END
```

The PF function on CUSTNO causes the first position to be isolated for testing on the IF command.

This example shows the partial field function being used where a field name or constant could be used as a part of a logical expression. The PF function represents a value to be tested against another value. In the case shown, the condition will be true.

Example 3

The field VARIABLE is a variable length field.

Field:	VARIABLE	PF(VARIABLE,3)	PF(VARIABLE,2,4)
Field contents:	'ABCDEFGHIJ'	'CDEFGHIJ'	'BCDE'

The value of the partial field is the part of field VARIABLE defined by a starting position and a length parameter. For variable length fields, you can omit the length parameter to signify the remainder of the field.

Chapter

4

Procedure Statements

The following list summarizes the ASL commands for procedure statements. Longer command and keyword names have abbreviations that are defined below the command or keyword name.

CALL	Calls another procedure or invokes report or subfile output.
CASE	Begins a group of procedure statements within a DO CASE block that are to be performed when the CASE condition is true.
COLLATE	Specifies the reports that are to be collated and the length of the collate key field. **Note:** COLLATE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.
COMBINE **COM**	Concatenates two or more character strings into a 1-character field.
CONTINUE **CONT**	Continues by going to the end of the procedure.
DO	Either initiates a conditional loop within a procedure or begins a block of procedure statements containing groups of CASE commands. Only the first CASE block within the DO CASE block with a true condition is executed.
ELSE	Begins a group of statements within an IF or DO CASE block that are to be performed when all conditions are false.
END	Terminates a DO, IF, or DO CASE block.
FIELD **FLD**	Defines a temporary field within a procedure.
GO	Jumps to a specified statement within a procedure.
IF	Begins a block of procedure statements that are performed when the condition is true on the IF command.
LEAVE	Provides a way to exit a DO loop immediately.

LET	Sets the value of a field equal to the value of a field, constant, or arithmetic expression.
LOCATE **LOC**	Locates a cell, row, or column of an array. **Note:** LOCATE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.
RELEASE **REL**	Releases a segment or an array occurrence.
REPLACE **REP**	Replaces defined characters within a character field with other defined characters.
RETURN **RET**	Returns control to the calling procedure.
ROUTE	Specifies the reports, selection criteria, if any, and destinations to which the reports are to be routed. **Note:** ROUTE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.
TRANSFER	Causes control to be transferred to a specific point within the processing cycle.

Syntax and Examples

This section describes each statement in detail. After the syntax and operands of a command are explained, several examples of the command are given. Values for fields are given, as well as other pertinent information and the result of the procedure statements.

Quotation Marks

When field values are shown, the contents of the field are enclosed in single quotation marks. For example, the contents of the field CUSTNO are enclosed in single quotation marks, but the quotation marks are not part of the field.

```
Field:      CUSTNO
Contents:   '00001'
```

The quotation marks delimit the beginning and end of the field.

For the rules and explanations of notational conventions and page layout, see Chapter 2, "Terminology, Syntax, and Processing".

CALL Command

```
CALL { [PROCEDURE] procedure-name |
       REPORT report-name |
       SUBFILE subfile-name |
       MODULE 'module-name' |
       CEEDATE | CEEDATM | CEEDAYS | CEEDYWK | CEEGMT |
       CEEGMTO | CEEISEC | CEELOCT | CEEQCEN | CEESCEN |
       CEESECI | CEESECS | CEEUTC
       [USING parm ...] }
```

Use the CALL statement to branch to subprocesses such as other internal procedures, external user written modules, reports, and subfiles. When the subprocess, invoked by the CALL command completes, the subprocess passes control back to the procedure statement following the CALL command.

The keywords beginning with CEE designate selected IBM® Language Environment® (LE) services that the CALL command specifically recognizes and assists with parameter conversion where necessary. Specifically, the CALL command automatically converts any parameter expected to be a length-prefixed character string (VSTRING) for any of these CEE... services whenever the CEE... keywords are used.

Note:

- The IBM Language Environment (LE) services, CEE..., are Year 2000 compliant.
- See the IBM *Language Environment for OS/390 & VM Programming Reference* manual for syntax and examples of these callable services.

LE services can also be invoked using the MODULE keyword. However, no parameter conversion takes place when this form is used.

PROCEDURE *procedure-name*
PROC

Specifies a procedure to be executed. Use the name of a procedure within the current application as the operand.

REPORT *report-name*
REP

Specifies that a report is to be output. Use the name of a report in the current application as the operand.

SUBFILE *subfile-name*
SUB

Specifies that data is to be output to a subfile. Use the name of a subfile in the current application as the operand.

MODULE *module-name*
MOD

Specifies an external user written load module to call. Use an external load module name as the operand (module-name).

CEEDATE	LE service that converts dates in the Lilian format to character values.
CEEDATM	LE service that converts number of seconds to character timestamp.
CEEDAYS	LE service that converts character date values to the Lilian format. Day one is 15 October 1582, and the value is incremented by one for each subsequent day.
CEEDYWK	LE service that provides day of week calculation.
CEEGMT	LE service that gets current Greenwich Mean Time (date and time).
CEEGMTO	LE service that gets difference between Greenwich Mean Time and local time.
CEEISEC	LE service that converts binary year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 15 October 1582.
CEELOCT	LE service that retrieves current date and time.
CEEQCEN	LE service that queries the century window.
CEESCEN	LE service that sets the century window.
CEESECI	LE service that converts a number representing the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond.
CEESECS	LE service that converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582.
CEEUTC	LE service that performs the same function as CEEGMT.

USING *parm...*

Specifies any parameters to be passed to the MODULE or CEE... service that was named in the previous keyword phrase.

Use field names or constants as the operand. Except as noted above, use parameter data types that conform to the data types expected by the called program.

Example 1

```
CALL   PROCEDURE   AVGSAL
```

This CALL command calls a procedure named AVGSAL, which is defined as a subroutine procedure. After AVGSAL executes, it passes control to the statement after the CALL to the subroutine procedure.

Example 2

```
CALL   REPORT   NAMELIST
```

This CALL command calls for output to a report named NAMELIST before continuing within the procedure.

Example 3

```
CALL  SUBFILE  MFDATA
```

This CALL command calls for a record to be output to the subfile MFDATA before continuing in the procedure.

Example 4

```
CALL  MODULE   MYPROG    USING   AMOUNT   FIELDX   'ABC'
```

This CALL command calls for an external module of code named MYPROG to be executed. The following parameters will be passed.

AMOUNT contains: `'10'`

FIELDX contains: `'011501'`

A literal value: `'ABC'`

The first parameter is the contents of field AMOUNT, which is 10; the second parameter is the contents of FIELDX, which is 011598; and the third parameter is the literal constant ABC.

Example 5

```
CALL CEEDAYS USING BRTHDATE 'MM/DD/YY' T.LILDATE T.FEEDBACK
```

This CALL command converts the date in the field BRTHDATE to a Lilian date. The string 'MM/DD/YY' is a picture string representing the format of the contents of the BRTHDATE field.

Example 6

```
CALL CEEDATE USING T.LILDATE 'DD Mmm YYYY' T.NEWDATE T.FEEDBACK
```

This CALL command converts the Lilian date in temporary field LILDATE into character format in the temporary field NEWDATE. The picture yields a date such as 15 Jan 2001.

CASE Command

CASE [WHERE] *logical-expression*

The CASE command begins a group of procedure statements within a DO CASE block. The procedure statements are performed when the condition of the CASE command is true.

The program performs only the first group of procedure statements whose CASE command condition is true within the DO CASE block. Once the group of procedure statements executes, control is transferred immediately to the END command corresponding to the DO CASE block.

WHERE *logical-expression* Specifies the logical expression to be evaluated to determine whether the following group of statements is to be performed.

If the logical expression is true, control passes to the statement immediately following the CASE command; otherwise, control passes to the next CASE, ELSE, or END command within the current DO CASE block.

Example

```
LET  T.TEMP1  =  0
DO  CASE
   CASE  WHERE  CUSTNO  EQ   '00001'
      LET  T.TEMP1  =  1
   CASE  WHERE  CUSTNO  GT  '00001'
      LET  T.TEMP1  =  2
  ELSE
     LET  T.TEMP  1  =  -1
END
```

This DO CASE block sets the value of T.TEMP1 to:

1 if CUSTNO is 00001.

2 if CUSTNO is greater than 00001.

-1 for all other cases (that is, CUSTNO is less than 00001).

COLLATE Command

Note: COLLATE is available for *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

COLLATE {REPORTS *report-name ...*
[KEYLENGTH *length*] | ALL KEYLENGTH *length* }

The COLLATE command specifies:

- The reports that are to be collated.
- The length of the collate key field.

You can use any number of COLLATE statements in an application. You can specify any given report in only one COLLATE statement.

REPORTS *report-name ...*

Specifies one or more reports that are to be collated and that become the members of this collection of reports. The report-name operand is a list of one or more request names (report names in VISION:Workbench for DOS).

If the request contains more than one report, identify specific reports within the request by stating the request name, a colon, and the specific report number(s) (Rn statement sets). For example, SALESRPT:513 indicates that the procedure will process only reports 1, 3, and 5 of the request SALESRPT for this COLLATE command, and that it will print report 5 before reports 1 and 3.

Although no embedded blanks are allowed in the list of report numbers, the rule concerning embedded blanks does not apply to this platform, because each report in VISION:Workbench for DOS is a uniquely named object.

The order of the request names and report numbers within requests determines the order of the final report output. Use the COLLATE command to re-order the report output sequence.

ALL

Specifies that all reports in the application are to be collated according to the data values contained in the collating key whose length is specified using the KEYLENGTH parameter. When you specify the ALL keyword, the KEYLENGTH parameter is required and only one COLLATE statement is allowed in the application.

KEYLENGTH *length*
KEYLEN

Specifies the size of the collating key, beginning at the first byte of the report key, used to control the collation of the report data.

- Do not specify a length that exceeds the length of the report key. The procedure groups all report data for the reports identified with the REPORTS keyword by the values in the collating key.
- When you do not specify the KEYLENGTH keyword in combination with the REPORTS keyword, the COLLATE statement prints the full reports in the order listed within the REPORTS keyword. No collation of report subsets from the different reports takes place.

Example

```
COLLATE REPORTS REP1 REP2 KEYLENGTH 1
```

REP1 and REP2 collate according to the collating key which is 1 byte in length. REP1 will print before REP2.

COMBINE Command

COMBINE [FIELDS] *field1 ...* STORE *result-field* [[BLANKS] *number*]

COMBINE
COM

The COMBINE command concatenates two or more character strings and store the result in a specified destination field. You can also specify the number of blanks inserted between each string.

FIELDS *field1*
FLDS

Specifies a list of character string fields. Make operands the names of character string fields (fixed or variable length) or literals enclosed in quotation marks as operands.

STORE *result-field*

Specifies the result field. Make the operand the name of a character string field (fixed or variable length) to contain the concatenation of all the literals and fields named in the FIELDS keyword phrase.

BLANKS *number*

Specifies the number of blanks to insert between each pair of strings. Use an integer constant as an operand. If you omit this keyword phrase, the default is zero (no blanks between fields).

Example 1

```
COMBINE  '('  AREA  ')'  PHONE  STORE  RESULT
```

For the following field values, the COMBINE concatenates the literal '(' to the field AREA with no spaces in between:

Fields	AREA	PHONE	RESULT
Contents before:	'213'	'555-1212 '	'9999999999999'
Contents after:	'213'	'555-1212'	'(213)555-1212'

Then, the literal ')' is concatenated, followed by the contents of the PHONE field. The result of the concatenations is stored in field RESULT.

Example 2

```
COMBINE FIELDS PF(AREA 1 1) PF(AREA 2 1) PF(AREA 3 1) ,
     STORE T.AREA BLANKS 1
```

For the following field values, the concatenation is done with one blank between each field:

Fields	AREA	T.AREA
Contents before:	'213'	'99999'
Contents after:	'213'	'2 1 3'

The partial field function is used on the fields in the FIELDS keyword phrase. The command is also coded on two lines using a comma at the end of the first line to denote a continuation of the procedure statement.

CONTINUE Command

CONTINUE
CONT

The CONTINUE command causes the normal record processing cycle to continue with the next occurrence of the segment within the controlling set of segments. (See Implicit Loops and Set Operation in Chapter 2, "Terminology, Syntax, and Processing".)

Note that this action is equivalent to a RETURN command if this procedure has been initiated by another procedure and there are no more occurrences of the segment.

Example 1

```
IF  CUSTNO  EQ  '00001'
     LET  T.COUNT = T.COUNT  +  1
     CALL  REPORT  LISTX
ELSE
     CONTINUE
END
CALL  SUBFILE  SUBF1
```

A CUSTNO value of 00001 causes the LET command to execute followed by the CALL command to the LISTX report. Then, the CALL command to the subfile SUBF1 is executed.

Any other value for CUSTNO causes the CONTINUE command to execute, and the CALL command to the subfile SUBF1 is not executed.

Example 2

```
IF CUSTNO LT '10000'
      CALL  REPORT  FIRST
  CONTINUE
END
CALL REPORT SECOND
```

A value of CUSTNO less than 10000 causes the CALL command to report FIRST to execute. The CONTINUE bypasses the CALL command to report SECOND. A value of CUSTNO greater than or equal to 10000 causes the CALL command to report SECOND to execute.

DO Command

```
DO  {[WHILE logical-expression]
     [UNTIL logical-expression]
     [FORALL segment-name]
    {[FORALL CELLS IN ARRAY array-identifier]
     [FORALL COLUMNS IN ARRAY array-identifier
               (WITHIN ROW row-number]]
     [FORALL ROWS IN ARRAY array-identifier
               (WITHIN COLUMN column-number]]}
     [FOR integer] } |
     [CASE]
```

Array keywords are available in VISION:Builder and VISION:Two, but not VISION:Inform.

The DO command begins a group of procedure statements called a DO block. The procedure statements within the DO block define an explicit procedural looping structure, the exception being individual CASE blocks.

Note that implicit looping (see Chapter 2, "Terminology, Syntax, and Processing"), still occurs within DO CASE blocks.

The keyword phrases that follow the DO command determine which type of processing is performed. The DO block must have a corresponding END command.

WHILE *logical-expression*

Specifies a logical expression that is evaluated at the beginning of each iteration of the explicit loop (that is, at the DO command).

- If the condition is true, the programs performs another iteration of the explicit loop performed.
- Otherwise, the programs terminates the explicit loop and transfers control to the statement after the END command associated with the DO block.

Change the conditions in the logical expression within the DO block in order to terminate the explicit loop.

Do not use the WHILE keyword phrase with the CASE keyword.

UNTIL *logical-expression*

Specifies a logical expression that is evaluated at the end of each iteration of the explicit loop (that is, at the END command).

- If the logical expression is true, the program terminates the explicit loop and transfers control to the statement after the END command associated with the DO block.
- Otherwise, the programs performs another iteration of the explicit loop.

Change the conditions in the logical expression within the DO block in order to terminate the explicit loop.

Do not use the UNTIL keyword phrase with the CASE keyword.

FORALL *segment-name*

Specifies the name of a segment.

- If the segment is a root segment from a user-read additional file, the program reads a record from the file at each iteration of the DO block. The processing of the DO block finishes at the end of all occurrences of the segment for the current record.
- If the segment is a lower level segment, the programs executes the DO block for all of the segment occurrences.

Each iteration of the DO block processes the next occurrence of the specified segment.

Do not use the FORALL keyword phrase with the CASE keyword.

ARRAY *array-identifier*
ARR

Specifies the array-identifier for the FORALL keyword. Use the name of an array in the current application as the operand.

ROW *row-number*

Specifies the row of the array for the FORALL keyword. Use a constant, field name, or arithmetic expression, whose value specifies the row number to locate, as the operand.

COLUMN *column-number*
COL

Specifies the column of the array for the FORALL keyword. Use a constant, field name, or arithmetic expression, whose value specifies the column number to locate, as the operand.

FOR *integer*

Specifies the maximum number of times the program executes the DO block. Use an integer value from 1 to 99999.

If the conditions set up under the WHILE or UNTIL operands have not been met within the specified number of interactions on the FOR keyword phrase, the program automatically terminates the DO block at the maximum number.

Do not use the FOR keyword phrase with the CASE keyword.

CASE

Specifies that CASE command blocks follow. This keyword has no operands. You can have an ELSE command in a DO CASE block.

- The program executes the first CASE command whose conditions are met within the DO CASE block and passes control to the corresponding END command.
- The CASE operand does not cause any explicit loop processing like the WHILE, UNTIL, and FORALL keyword phrases. However, all implicit looping (see Chapter 2, "Terminology, Syntax, and Processing"), still occurs.

Do not use the CASE keyword with the WHILE, UNTIL, FORALL, or FOR keywords.

If the condition in the WHILE operand is initially false, the program does not execute the DO block. Conversely, the UNTIL keyword makes it possible for the program to execute the statements within the DO block at least once even though the condition is initially true.

You can use the WHILE, UNTIL, FORALL, and FOR keywords in combination with each other.

If you specify both the FORALL and WHILE keywords, the program retrieves the first occurrence of the segment before the WHILE condition is evaluated.

Example 1

```
LET T.COUNT = 0
DO WHILE T.COUNT =  0
  CALL REPORT SAMPLE
  LET T.COUNT = 1
END
```

In the previous example:

- The program sets the field T.COUNT to zero before starting the DO block; hence the condition in the DO command is true and the program executes the procedure statements within the DO block.
- In the second interaction of the explicit loop, the condition in the DO command fails and the program bypasses the procedure statements.
- The programs passes control to the next procedure statement after the END for the DO block.
- The program processes the CALL command within the DO block only once each time the procedure is executed.

Example 2

```
LET T.RECNUM = 0
DO UNTIL T.RECNUM EQ 3
  CALL SUBFILE RECOUT
  LET T.RECNUM = T.RECNUM + 1
END
```

Because the program sets T.RECNUM to zero before starting the DO block, the program executes the procedure statements within the DO block.

- At the end of the DO block, T.RECNUM contains a 1 and the UNTIL condition fails, allowing the DO block to be executed a second time.
- After the DO block executes a second time, T.RECNUM contains a 2 and the UNTIL condition fails again, allowing the DO block to be executed a third time.
- After the DO block executes a third time, T.RECNUM contains a 3 and the UNTIL condition is true, causing the DO UNTIL to be satisfied.

Processing continues with the procedure statement after the END command corresponding to the current DO command.

Example 3

```
LET T.FIELD1 =  0
DO FORALL  1.ITEM
  LET  T.FIELD1 =  1.FIELD1 + T.FIELD1
END
```

The segment 1.ITEM is a root segment from a user-read additional file. The program reads a record from the file at the beginning of the DO block.

- If there is a record in the file, the program executes the procedure statements within the DO block.
- If there are no records to read, the DO block fails and the program bypasses the procedure statements in the DO block. The program reads another record for each execution of the loop and the previous record is no longer available to process.

The program executes the DO block once for each record in file 1. Therefore, the field T.FIELD1 accumulates a total of all of the values of 1.FIELD1 from each record of file 1. The programs terminates the DO block at the end of file on file 1 and then executes the statement following the END command.

For all files read with the DO FORALL, the program reads the file until EOF is reached. The last record read is not kept for processing after the DO block.

Example 4

```
LET T.AMOUNT = 0
DO FORALL ROWS IN ARRAY A.XYZ WITHIN COLUMN T.COLNO,
                              FOR 50,
                              WHILE A.CLASS = 'X',
                              UNTIL T.AMOUNT > 5000
   LET T.AMOUNT =  T.AMOUNT + A.AMOUNT
END
```

This example loops through up to 50 rows of column T.COLNO as long as A.CLASS contains an X and until T.AMOUNT exceeds 5000.

Example 5

```
DO CASE
 CASE CLASS = 1
  LET RATE = RATE * 1.05
 CASE CLASS = 2
  LET RATE = RATE * 1.10
END
```

For the following field values, the program evaluates the first CASE command and the logical expression is false:

Field:	CLASS	RATE
Contents before:	'2'	'5.00'
Contents after:	'2'	'5.50'

The program evaluates the second CASE command and the logical expression is true. The programs executes the procedure statement following the second CASE command and passes control to the END command corresponding to the current DO CASE block.

The program executes the procedure statements following the first CASE command evaluated with a true logical expression and immediately passes control to the END command for the DO CASE block.

For the same code with the following field values, all of the logical expressions in the CASE commands are evaluated as false:

Field:	CLASS	RATE
Contents before:	'0'	'1'
Contents after:	'0'	'1'

Because an ELSE command is not coded, the program immediately passes control to the END command for the DO block.

ELSE Command

ELSE

The ELSE command begins a group of procedure statements within an IF or DO CASE block that execute if the conditions in the preceding IF command, or all of the preceding CASE commands, are false. The ELSE command within a DO CASE block must follow all of the CASE commands within the same DO CASE block. There are no keywords.

Example 1

```
IF PF (CUSTPH 1 3) =  '713'
  LET T.TITLE =  'HOUSTON'
ELSE
  LET T.TITLE = 'OUTSIDE OF HOUSTON'
END
```

For the following field values, the logical expression in the IF command is false:

Field:	PF (CUSTPH 1 3)	T.TITLE
Contents before:	'409'	'ZZZZZZZZZZZZZZZZZZ'
Contents after:	'409'	'OUTSIDE OF HOUSTON'

Because an ELSE command is coded, the program transfers control to the ELSE command and executes the procedure statements that follow.

For the same code with the following field values, the logical expression in the IF command is true:

	PF (CUSTPH 1 3)	T.TITLE
Contents before:	'713'	'OUTSIDE OF HOUSTON'
Contents after:	'713'	'HOUSTON '

The program processes the procedure statements following the IF command until it encounters the ELSE command. The program then passes control to the corresponding END command for the current IF command.

Example 2

```
DO CASE
      CASE AREACODE= '713'
      LET T.TITLE= 'HOUSTON'
      CASE AREACODE= '806'
      LET T.TITLE= 'NORTH WEST TEXAS'
      CASE AREACODE= '915'
      LET T.TITLE= 'WEST TEXAS'
      CASE AREACODE= '512'
      LET T.TITLE= 'SOUTH WEST TEXAS'
      CASE AREACODE= '409'
      LET T.TITLE= 'S.E. TEXAS (EXCEPT HOUSTON)'
      CASE AREACODE= '817'
      LET T.TITLE= 'NORTH CENTRAL TEXAS (FT. WORTH)'
      CASE AREACODE= '214'
      LET T.TITLE= 'N.E. TEXAS (INCLUDING DALLAS)'
       ELSE
      LET T.TITLE= 'OUTSIDE OF TEXAS'
END
```

If AREACODE is one of the Texas state area codes, the program assigns T.TITLE field the portion of the state corresponding to the area code. If the area code is not one of the Texas state area codes, the program executes the ELSE command and sets the T.TITLE field to OUTSIDE OF TEXAS.

END Command

END

The END command signifies the end of an IF or DO block. The END command has no keywords.

Example.

```
DO FORALL 1.PARM
 IF 1.RECORD NE ' '  THEN
  LET  T.FIELD1 = 1.FIELD1
  LET  T.FIELD2 = 1.FIELD2
  LET  T.FIELD3 = 1.FIELD3
 END
END
```

```
 DO CASE
  CASE CLASS = 1
   LET RATE = RATE * 1.05
  CASE CLASS =  2
   LET RATE = RATE * 1.10
 END
```

Specify an END command for every DO or IF command.

In the example, the first END command delimits the end of the embedded IF command. The second END command delimits the end of the DO FORALL command. The third END command delimits the end of the DO CASE command.

FIELD Command

***field name*:FIELD [TYPE] *field type* [[LENGTH] *field-length*]**
[DECIMALS *decimal-places*]
[FLOAT *floating-edit-char*]
[FILL *fill-edit-char*]
[TRAIL *trailing-edit-char*]
[EDLEN *edit-length*]
[INIT *initial-value*]
[HEADING `*line1*` [`*line2*`]]

The FIELD command defines a temporary field within a procedure. Once you define a temporary field in the FIELD command, you can use it in other procedures, reports, or subfiles. Make it a practice to define all temporary fields in the first procedure in the application. The statement label is the field name.

***field name:* FIELD**
FLD

Specifies a temporary field name.

- Make field names from one to eight characters long starting with an alphabetic character. See Chapter 2, "Terminology, Syntax, and Processing" for valid field names.
- Place a colon at the end of the field name followed by one or more blanks.
- Even though the syntax appears to be "labeling" the line, the FIELD keyword causes the line to be a temporary field definition.
- Specify all FIELD statements at the beginning of a procedure.

TYPE *field-type*

Specifies the field type. Make this operand a 1-character code corresponding to the storage type of the field being defined (for example, C for character type field, Z for zoned decimal).

For valid entries, see Appendix B, "Technical Notes".

LENGTH *field-length*
LEN

Specifies the field length. Use an integer constant, with a value within the range of the limits imposed by the field type, as the operand.

For default field lengths for each field type, see Appendix B, "Technical Notes".

DECIMALS *decimal-places* **DEC**	Specifies the number of decimal places for numeric fields. Use an integer from 0 through 9 as the operand. Make this value smaller than the length of the numeric field.
FLOAT *floating-edit-char* **FLT**	Specifies the floating edit character. Use a single character constant as the operand. For valid floating edit characters, see Appendix B, "Technical Notes".
FILL fill-edit-char	Specifies filling edit character. Use a single character constant as the operand. For valid fill edit characters, see Appendix B, "Technical Notes".
TRAIL *trailing-edit-char* **TRL**	Specifies the trailing edit character. Use a single character constant as the operand. For valid trailing edit characters, see Appendix B, "Technical Notes".
EDLEN *edit-length*	Specifies an overriding edit length. Use an integer constant as the operand. For valid edit length entries, see Appendix B, "Technical Notes".
INIT *initial-value*	Specifies an initial value. Use a constant, whose type corresponds to the field type (For example, for numeric type fields use numeric digits, as the operand. Add a negative sign and decimal point, if needed).

HEADING *'line1' 'line2'*
HEAD

Specifies one or two lines of column heading for the field if it is output to a report.

- Use character constants enclosed in single quotation marks as the operands.
- Make each line of heading up to 14 characters long.
- To force a blank column heading, insert the system delimiter after a blank within the quotation marks.
- Use two consecutive single quotation marks within a heading to represent a single quotation mark.
- If you leave this operand blank, the field name is used as the column heading.

Example 1

```
FRSTTIME:  FIELD TYPE  C  LENGTH  1  INIT 'Y'
```

You can reference the temporary field by using the qualifier T in front of the field name (for example, T.FRSTTIME). The field contains character type values (such as A to Z, 0 to 9, blanks) and is one character long with an initial value of Y.

Example 2

```
BALANCE: FIELD Z 8 2 HEADING 'CUSTOMER' 'BALANCE'
```

The temporary field T.BALANCE contains numeric values (0 to 9, positive or negative), has two decimal places, and is eight digits long. The first six digits are to the left of the implied decimal place. If the temporary field is output to a report, the column heading:

```
CUSTOMER
BALANCE
```

is centered over the column of data being printed.

GO TO Command

GO [TO] *jump-to-label*

The GO command branches forward to another labeled procedure statement in the current procedure. Create labels for procedure statements by starting the line with the label followed immediately by a colon and one or more blanks.

TO *jump-to-label* Specifies the procedure statement to be executed next. You can only jump to a subsequent labeled statement in the current procedure.

The colon used on the labeled procedure statement is not used on the GO TO command.

The GO command only branches forward to a subsequent labeled statement in the current procedure. Do not attempt to use the GO command to branch backward to a prior labeled statement.

You can use a GO statement to jump out of an IF-END or DO-END group of statements. If you use a GO in this manner, it overrides the WHILE, UNTIL, FOR, and FORALL conditions governing the DO loop.

Note: The periods mean that some of the procedure code has been omitted from the example.

Example 1

```
.
.
.
GO TO LABEL10
.
.
.
LABEL10: LET A = A + 1
.
.
.
```

The example above shows the syntax of the GO command branching to a labeled line forward in the procedure called LABEL10. The format of a label is shown on the procedure statement with the LET command. When the GO command is executed, all procedure statements between the GO command and the line labeled LABEL10 are bypassed and processing continues with the LET command on the labeled line.

Note: The periods between the END command and the ERROR labeled statement mean that some of the procedure code has been omitted from the example.

Example 2

```
IF VALIDATE(ORDRDATE  DATE)
 CALL REPORT ORDER
ELSE
 LET T.ERROR = ORDRDATE
 LET T.MSG = 'INVALID ORDER DATE'
 GO TO ERROR
END
;
IF VALIDATE(ITEMNO PATTERN P'9999999')
 CALL REPORT ITEM
ELSE
 LET T.ERROR = ITEMNO
 LET T.MSG = 'INVALID ITEM NUMBER'
 GO TO ERROR
END
.
.
.
ERROR: CALL REPORT ERROR
```

The label ERROR is in a CALL command for an error report. Only the first error detected is reported with this code.

- If the order date is an invalid date, T.ERROR and T.MSG are set to indicate the invalid date and a corresponding message. The next procedure statement is a GO TO command that branches forward in the procedure to the line labeled ERROR at the bottom of the procedure.
- If the order date is valid, the CALL command for the report ORDER is executed, and processing continues with the next IF command. If the field ITEMNO does not contain all numerics according to the pattern, T.ERROR and T.MSG are set to indicate the invalid item number and a corresponding message. The next procedure statement is a GO TO command that branches forward in the procedure to the line labeled ERROR at the bottom of the procedure.

IF Command

IF [CONDITION] *logical-expression* [THEN]

Place the IF command at the beginning of a group of statements (an IF block) to be performed when the logical expression on the IF command is true.

The program transfers control to the corresponding ELSE or END statement for this IF block when the logical expression is false.

Use an END command to delimit the IF block of procedure statements.

CONDITION *logical-expression*
COND

Specifies the logical expression. The logical expression can contain:

- Arithmetic sub-expressions (see examples for further explanation).
- PF, LOOKUP, FIND, SCAN, and VALIDATE functions.

THEN

Is an optional entry with no operands. Use it for clarity and readability.

Example 1

```
IF CUSTNO = '00001' THEN
  LET T.SET = 'ON'
ELSE
  LET T.SET = 'OFF'
END
```

When the field CUSTNO contains 00002, the programs evaluates the logical expression on the IF command as false, passes control to the ELSE command within the IF block, and sets the field T.SET to OFF.

Example 2

```
IF PF(ORDRDATE 5 2) = PF(F.TODAY 5 2) -1 THEN
  CALL REPORT LASTYEAR
END
```

If the field ORDRDATE contains 011594 and the field F.TODAY contains 011595, the logical expression on the IF command is true.

The logical expression on the IF command has an arithmetic sub-expression PF(F.TODAY,5,2) -1. The last two positions of the flag field TODAY minus 1 are compared to the last two positions of ORDRDATE. This logic isolates the year portions of the two fields to see if the order was made last year with respect to the current year.

Example 3

```
IF SCAN(CUSTNAME FOR 'ACME') THEN
  CALL REPORT COMPANY
END
```

The SCAN function serves as the logical expression, because it is a conditional function. The program searches for the characters ACME in the field CUSTNAME.

- If the programs finds the character string, the SCAN function is true and the programs executes the CALL command for the report COMPANY.
- Otherwise, if the logical expression is false, the program bypasses the procedure statements between the IF command and the END command and passes control to the statement following the END command.

LEAVE Command

LEAVE

The LEAVE command exits the corresponding DO block immediately (that is, transfers control to the statement after the END command for the current block).

If the LEAVE command is within a nested DO block, the exit is to the outer DO block. See Example 3 on the following page.

The LEAVE command can only be used in DO blocks. The LEAVE command cannot be used in DO blocks with the keyword of CASE. The LEAVE command has no keywords.

Example 1

```
LET T.COUNT = 0
DO UNTIL T.COUNT = 1000
  IF FIND(SEGMENT 1.ROOT)
    LET T.TOTAL = T.TOTAL + 1.AMOUNT
    LET T.COUNT = T.COUNT + 1
  ELSE
    LEAVE
  END
END
```

In this example, the LEAVE command transfers control out of the DO block if there are less than 1000 records on the file being read by the FIND function. If the file being read on the FIND function runs out of records before the DO UNTIL command is satisfied, the false condition in the IF command passes control to the ELSE command where the LEAVE command is executed. Without the LEAVE command, there would be no way to exit the DO-END set of statements.

Example 2

```
LET  T.ORDRCNT = 0
DO FORALL ORDER
  IF ORDCMPLT = 'Y' THEN
    LET T.ORDRCNT = T.ORDRCNT + 1
  ELSE
    CALL REPORT ERROR
    LEAVE
  END
END
```

This section of code counts all of the completed orders. If any order is not complete (that is, ORDCMPLT is not Y), the program executes the ELSE command along with the subsequent procedure statements. The program processes the report ERROR and exits the DO block without processing all of the ORDER segments.

Example 3

```
DO UNTIL T.COUNT = 1000
  DO FORALL SEG20
  .
  .
  .
    LEAVE
  .
  .
  .
  END
END
```

The LEAVE command is within the inner DO block and when it is executed, the program passes control to the statement after the END command which corresponds to the inner DO block. The outer DO block is not affected by the LEAVE command.

LET Command

LET [FIELD] *result-field = source-expression* [WITH][EDIT P'*pattern*'] [ROUNDING] [JUSTIFY {LEFT | RIGHT}]

Note: The IBM Language Environment (LE) service, CEEDATE, is Year 2000 compliant.

The LET command replaces or moves data between the source-expression and the result-field.

FIELD *result-field* FLD	Specifies the receiving field. Use a field name or PF function as the operand.
source-expression	Specifies the "from" field or arithmetic expression. Use a name, constant, or arithmetic expression, that can include any value function (PF or LOOKUP), as the operand.
WITH	Is an optional entry with no operands. Use it for clarity and readability where needed.
EDIT P'*pattern*'	Specifies the edit pattern that is used to edit the result-field. Specify a pattern string starting with the letter P, followed by a beginning single quotation mark, a string of special pattern symbols, and ending with a closing quotation mark. It can be up to 31 characters long. Use EDIT only if the result-field is a character string. For valid edit pattern symbols, see Appendix B, "Technical Notes". If the source-expression is a date field, the edit pattern is a date format picture, for example MM/DD/YY. For a list of valid picture characters. Use the date format picture as a parameter in a call to CEEDATE (see the *IBM Language Environment Reference Manual)*.
ROUNDING	Specifies that the result-field for all arithmetic and replacement operations should contain rounded results. Note that with the possible exception of divide, no rounding occurs if the number of decimal places in the result-field is not less than the number of decimal places in the computed or replacement value.

JUSTIFY {LEFT | RIGHT} Specifies that the result is to be left-justified or right-justified after the move of the data. The direction operand must be LEFT or RIGHT to indicate the direction in which the result-field is to be justified.

- If you omit RIGHT or LEFT, no action beyond replace is taken.
- JUSTIFY can only be used if the result-field is a character string.
- If EDIT and JUSTIFY are both specified, the JUSTIFY take places after the edit.

Example 1

```
LET  T.NUMBER  =  1.5
```

For the following field values, the temporary field T.NUMBER is replaced with the numeric constant on the right side of the equal sign:

Field:	T.NUMBER
Contents before:	'0.0'
Contents after:	'1.5'

Example 2

```
LET FIELD1 = FIELD2 JUSTIFY LEFT
```

For the following field values, the contents of field FIELD2 are moved into the field FIELD1:

Field:	FIELD1	FIELD2
Contents before:	'ZZZZZ'	' ABC'
Contents after:	'ABC '	' ABC'

The keyword JUSTIFY specifies LEFT for the direction and left-justifies the contents of the FIELD1 field at the end of the LET command.

Example 3

```
LET  A  =  X*Y+10
```

For the following field values, the contents of field X are multiplied by the contents of field Y, 10 is added, and the result is stored in field A:

Field:	A	X	Y
Contents before:	'0'	'5'	'5'
Contents after:	'35'	'5'	'5'

Example 4

```
LET T.AMOUNT = PAY + 100 WITH EDIT P'$,$$$.99'
```

For the following field values, the contents of field PAY plus 100 are stored in the field T.AMOUNT with the specified edit pattern:

Field:	T.AMOUNT	PAY
Contents before:	'0'	'950.25'
Contents after:	'$1,050.25'	'950.25'

Example 5

```
LET  RESULT  =  LOOKUP(MONTHS  ARGUMENT  ORDERMM)
```

This example uses the following lookup table where a number represents a month.

Months	(table)
01	JANUARY
02	FEBRUARY
03	MARCH
04	APRIL
05	MAY
06	JUNE
07	JULY
08	AUGUST
09	SEPTEMBER

Months	(table)
10	OCTOBER
11	NOVEMBER
12	DECEMBER

```
          RESULT                ORDERMM
Before:   'ZZZZZZZZZ'           '05'
After:    'MAY       '          '05'
```

The program "looked up" value of the field ORDERMM in the argument list of the MONTHS table. When the program finds a match, the program uses the result value from the table as the value of the LOOKUP function and, in this case, puts it into the field RESULT.

Example 6

```
LET PF(T.STRING 02 04) = T.MIDDLE
```

For the following field values, the LET command moves the contents of the field T.MIDDLE into the field T.STRING starting in location 2 for a length of 4:

```
Field:              T.STRING        T.MIDDLE
Contents before:    '123456789'     'ABCD'
Contents after:     '1ABCD6789'     'ABCD'
```

LOCATE Command

Note: LOCATE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

LOCATE [ARRAY] *array-identifier*
{[ROW *row-number*][COLUMN *column-number*]}

LOCATE **LOC**	This command processes arrays by locating a particular data cell in the array, a particular row in the array, or a particular column in the array.
ARRAY *array-identifier* **ARR**	Specifies the array-identifier. Use an identifier that identifies an array in the current application as the operand.
ROW *row-number*	Specifies the row of the array. Use a constant, field name, or arithmetic expression, whose value specifies the row number to locate, as the operand.
COLUMN *column-number* **COL**	Specifies the column of the array. Use a constant, field name, or arithmetic expression, whose value specifies the column number to locate, as the operand.

- If you use both ROW and COLUMN, the program locates a specific data cell in the array.
- If you code only ROW, the program processes all data cells in the row specified.
- If you code only COLUMN, the program processes all data cells in the column specified.

The program sets the system flag F.COLUMN to the column number being processed as a result of the LOCATE command and sets the system flag F.ROW to the row number being processed.

Array-identifier: A

Array name: BIRTHDAY

	COLUMN 1	COLUMN 2
ROW1	JOE 020359	KAREN 122551
ROW2	DOUG 050658	SARA 100960

Example 1

```
LOCATE ARRAY A ROW  2  COLUMN  1
```

The program finds the data cell with DOUG and 050658 as a result of the LOCATE command. When you use the field name(s) for the fields in the data cell, the program processes only the one data cell that is located. At the end of the procedure, the program automatically releases the array. The RELEASE command can be performed before the end of the procedure to allow processing of the entire array again.

Example 2

```
LOCATE A ROW X+1
```

If the field X contains 1, the arithmetic expression results in row 2 being located (F.ROW contains 2). When fields in the array are referenced after the LOCATE command, only row 2 is processed, looping through all columns in turn. F.COLUMN contains each column number as the column is being processed.

Example 3

```
LOCATE A COLUMN T.COL
```

If the field T.COL contains 1, the value of the temporary field results in column 1 being located (F.COLUMN contains 1). When fields in the array are referenced after the LOCATE command, only column 1 is processed, looping through all rows in turn. F.ROW contains each row number as the row is being processed.

RELEASE Command

RELEASE {[SEGMENT] *segment-name* | ARRAY *array-identifier*}

RELEASE
REL

Releases a segment specified by the FIND function or an array specified by the LOCATE command.

A FIND or LOCATE command limits the application's view of data that is available. Subsequent field references process data only from segments that result from a FIND command or cells that result from a LOCATE command.

The RELEASE command unlocks the application view and makes available all segments, or cells, to subsequent statements.

After the RELEASE command in the procedure, any reference to the segment or array results in processing, beginning with the first occurrence of the segment or the beginning of the array.

Note: The RELEASE command ARRAY keyword is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

SEGMENT *segment-name*
SEG

Specifies a segment name to release. Do not use the name of a root segment as the operand.

ARRAY *array-identifier*
ARR

Specifies the identifier of an array to release. Use an identifier corresponding to the array to be released as the operand.

Example 1

```
IF FIND(SEGMENT ORDER WHERE ORDERNO = 10001)
 RELEASE SEGMENT ORDER
 CALL REPORT ORDERS
END
```

The FIND function is true if there is at least one segment with an order number of 10001. The function positions on the first segment that is found.

The RELEASE command releases the ORDER segment that was found and the report outputs all of the current orders. Without the RELEASE command, the report ORDERS would only have access to the single segment established by the FIND function.

Example 2

The Birthday array from the examples for the LOCATE command is being used in this example.

```
LOCATE ARRAY A ROW 1 COLUMN 1
IF A.NAME EQ 'JOE 020359' THEN
  CALL REPORT NAMES
ELSE
  RELEASE ARRAY A
END
```

First, the program locates a cell of the array with the LOCATE command giving the ROW and COLUMN of the data cell. The program tests the field A.NAME in the array and, if the field does not contain JOE 020359, executes the ELSE command and releases the array. Otherwise, the data cell located at row 1 column 1 is the only cell available after the END of the IF block.

REPLACE Command

REPLACE [STRING] *search-string* [IN] *modify-field* [WITH] *substitute-value*

REPLACE
REP

This command replaces all occurrences in modify-field of search-string with substitute-value.

- The scan of the modify-field for the search-string begins at the leftmost character in the modify-field.
- The substitute-value is substituted into the modify-field in place of the search-string.
- The scan of the modify-field for the search-string then resumes starting with the first character after the substituted characters.

This process continues until the end of the scanned field is reached.

STRING *search-string*
STR

Specifies the string to search for in modify-field. The operand can be a field name, character constant, or validation pattern.

For valid pattern symbols, see Appendix B, "Technical Notes".

IN *modify-field*

Specifies the field to be modified (that is, the field that contains the search-string). Use a field name or a PF function as the operand.

WITH *substitute-value*

Specifies the value to be substituted into modify-field instead of search-string. Use a field name or character constant as the operand.

The system field SSCOUNT counts the number of matches found during the REPLACE operation.

- If the search-string is longer than the modify-field, the F.SSCOUNT system field is set to zero.
- If the modify-field or the search-string is null (an empty variable length field) or invalid and/or the substitute-value is invalid, the operation is not performed and the F.SSCOUNT system field is set to -2.
- If the search-string and the substitute-value are of different lengths, the REPLACE command is executed according to the following rules:
  - If the search-string is smaller than the substitute-value, the portion to the right of the matching characters is shifted to the right to make room for the characters to be substituted.

Non-blank characters might be lost on the right (truncated) by this process, unless the modify-field is a variable length field.

If the substitution would cause the maximum length of a variable length field to be exceeded, the operation does not take place, the modify-field becomes invalid, and the system field F.SSCOUNT is set to a -1.

– If the search-string is larger than the substitute-value, the portion of the modify-field to the right of the matching characters is left-justified and placed adjacent to the substituted characters.

  If the modify-field is a fixed length character field, trailing blanks are created in the field.

  If the substitute-value is a variable length field with a null length, a zero length field is substituted for the search-string, effectively eliminating it and left-justifying the remaining portion of the field.

Example 1

```
REPLACE 'ABC' IN CUSTNAME WITH 'XYZ'
```

For this example, each time the literal ABC is found in the field CUSTNAME, it is replaced with the literal XYZ. F.SSCOUNT is set to 1, because one match was found:

Field:	`CUSTNAME`
Contents before:	`'THE ABC COMPANY'`
Contents after:	`'THE XYZ COMPANY'`

Example 2

```
REPLACE P'ZZZ999' IN ITEMNO WITH '111QQQ'
```

For this example, each time the pattern of three alphabetics followed by three numerics is found in the field ITEMNO, it is replaced by the literal 111QQQ. F.SSCOUNT is set to 1 because one match was found:

Field:	`ITEMNO`
Contents before:	`'3AAA222C'`
Contents after:	`'3111QQQC'`

Example 3

```
REPLACE  '/'  IN  SHIPDATE  WITH  T.NULL
```

For this example, each time the program finds '/' in the field DATE, the program substitutes the null value from the field T.NULL. The program sets F.SSCOUNT to 2, because two matches were found:

Field:	SHIPDATE	T.NULL (TYPE V, EMPTY)
Contents before:	'01/15/01'	' '
Contents after:	'011501 '	' '

RETURN Command

RETURN
RET

Use the RETURN command in a subroutine procedure to branch back to the calling procedure. Note that a RETURN statement is not required for a subroutine procedure to get back to the calling procedure, because the end of a procedure constitutes an automatic RETURN.

Control automatically passes back to the calling procedure after all of the statements in the subroutine procedure process.

Example

```
IF ORDCMPLT = 'Y'
 CALL REPORT ORDERS
 RETURN
END
;
IF  DATE LE '010115'
 CALL REPORT TRACKDWN
 RETURN
END
CALL REPORT ALLOTHRS
```

These procedure statements make up a subroutine procedure that you can CALL from any other procedure. When the subroutine procedure executes the RETURN commands, the subroutine procedure passes control back to the calling procedure.

If ORDCMPLT is equal to Y or the DATE is less than or equal to 010115, The program produces a report line and immediately exits the subroutine procedure by one of the RETURN commands. The subroutine procedure automatically exits after processing the CALL command to report ALLOTHRS where the RETURN is implied.

ROUTE Command

Note: ROUTE is available in *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

ROUTE { REPORTS *report-name ...* | ALL } [KEYVALUE '*data-value*']
TO *destination ...* [DEFER]

The ROUTE command specifies the reports, selection criteria (if any) and destinations to which the reports are to be routed.

- You have many ROUTE statements in an application.
- You can include any report in multiple ROUTE statements.
- A unique ROUTE statement is required for each combination of reports, selection-data, and destinations.
- Reports that are not identified on any ROUTE statement are routed to the default destination.
- There are no ROUTE command dependencies with regard to the COLLATE command or vice versa.

Each of these commands stands alone in its function.

REPORTS *report-name ...*
REPORT

Specifies the reports that are to be routed to the designated destinations. The report-name operand is a list of one or more request names (report names in VISION:Workbench for DOS).

If the request contains more than one report and only selected reports within the request are to be routed to the designated destinations, the request name can optionally be followed by a colon, which is then followed by the report numbers (Rn statement sets) that are to be used.

For example, a specification of SALESRPT:135 indicates that only reports 1, 3, and 5 of the request SALESRPT are to be considered for this destination.

Although no embedded blanks are allowed in the list of report numbers, the rule concerning embedded blanks does not apply to this platform, because each report in VISION:Workbench for DOS is a uniquely named object.

ALL

Specifies that ALL reports in the application are to be routed to the designated destination(s).

KEYVALUE '*data-value*'
KEYVAL

Specifies a character constant (string value enclosed in single quotation marks) indicating that only the report data whose routing key value matches the specified data value are to be routed to the designated destinations. (The routing key is a subset of the report key beginning with position 1 of the report key.)

The length of the KEYVALUE keyword operand specifies the length of the routing key and can include trailing blanks. Make the length of this value less than the length of the report key.

TO *destination ...*

Specifies one or more destinations to which this report or set of reports is to be routed.

Destinations are specified as DTF/DD names that identify the operating system JCL statements, which in turn specify the appropriate data set destination parameters. When you use the special notation 'ddname (member-name)' (OS/390 (z/OS) systems only), the program assumes the destination to be a PDS and places the output in the member identified by member-name.

- The TO keyword is required with this statement.
- Do not use M4LIST as a report destination.

DEFER

Specifies that the OPEN command for the destination data set is to be deferred until just before the first output is to be written to the destination. This would then prevent the creation of empty data sets or unnecessary tape mounts when no data was selected for the report.

When you do not specify this keyword, the OPEN command for the destination data set will occur during program initialization regardless of whether any data was selected for the destination or not. If the programs does not select any data for the report, the program creates an empty data set.

Example

```
ROUTE REPORTS REP1 REP2   KEYVALUE    'A'    TO    COMMSALES
ROUTE REPORTS REP1 REP2   KEYVALUE    'B'    TO    GOVTSALES
ROUTE REPORTS REP1 REP2   KEYVALUE    'C'    TO    COMMSALES
ROUTE REPORTS REP1 REP2   KEYVALUE    'D'    TO    GOVTSALES
ROUTE REPORTS REP1 REP2   KEYVALUE    'A'    TO    BRANCH1
ROUTE REPORTS REP1 REP2   KEYVALUE    'B'    TO    BRANCH2
ROUTE ALL TO ARCHIVE
```

Reports REP1 and REP2 with a key value of:

A	are routed to	COMMSALES and BRANCH1
B	are routed to	GOVTSALES and BRANCH2
C	are routed to	COMMSALES only
D	are routed to	GOVTSALES only

Additionally, all reports are routed to ARCHIVE.

TRANSFER Command

TRANSFER [TO] {NEXT_MASTER | TYPE_1 | TYPE_2}

The TRANSFER command exits procedures immediately and transfers processing control to another type of procedure.

NEXT_MASTER Causes the system to read the next master file record and start the processing cycle again.

TYPE_1 Transfers control to the Transaction Record Initial Validation control procedure. Because this procedure has already been executed, a TRANSFER TYPE_1 constitutes recycling to an earlier point in the processing cycle. The results of all previous procedures (including the prior execution of the event-controlled procedure) are still in effect.

TYPE_2 Transfers control to the Transaction/Master File Alignment control procedure. Because this procedure has already been executed, a TRANSFER TYPE_2 constitutes recycling to an earlier point in the processing cycle. The results of all previous procedures (including the prior execution of the event-controlled procedure) are still in effect.

Example

```
IF CUSTOMER LT '01000'
 TRANSFER NEXT_MASTER
END
```

In this example, the program processes only CUSTOMER values greater than or equal to 01000 through the remainder of the procedure after the END command. For CUSTOMER values less than 01000, the TRANSFER command is executed. The program reads the next master file and the starts the processing cycle again.

Chapter

5

ASL Examples

This chapter contains examples of ASL that show how the language is used in the context of an entire procedure or application. The purpose of these examples is to illustrate how the language is used to perform a variety of operations that use the various features of the language.

Example 1

This example converts feet to centimeters. The program converts the value in field DISTANCE to centimeters and stores the result back into the field DISTANCE in the logical record. There is no change to the physical data.

```
; Convert distance in units of feet to distance in unit of
; centimeters.
;
LET DISTANCE = DISTANCE * 30.480 WITH ROUNDING
```

The WITH ROUNDING clause rounds the result to the nearest centimeter.

Because of the set operation of ASL, this one line procedure transforms all instances of the field DISTANCE in the database.

Example 2

This example reports only employee records for employees whose salaries are less than $50,000.

```
; Bypass all records for salaries of $50,000 and greater.
;
IF SALARY >= 50000
   TRANSFER TO NEXT_MASTER
END
```

All records for employees whose salaries are $50,000 or greater are ignored.

Example 3

This example reads a parameter record and places entries from the parameter record into working storage fields. If you do not provide a parameter record, the example writes the message text “NO PARAMETER RECORD” on the report ERRLIST and terminates the run by forcing end of file on the master file.

```
; Read parameter record into working storage.
;
IF FIND(9.PARMDATA)
   LET W.PARM1 = PF(9.PARMS 1 8)         ;First parameter field
```

```
   LET W.PARM2 = PF(9.PARMS 10 4)         ;Second parameter field
ELSE
   LET T.ERRMSG = 'NO PARAMETER RECORD'
   CALL REPORT ERRLIST                     ;Generate error report
   LET PF(F.EOF 1 1) = 'E'                ;Force End of File
END
```

The example:

- Reads the parameter record from file 9 using the FIND function.
- The FIND function returns a FALSE command if no record is available (that is, the file is empty or at end of file).
- Uses partial fielding of the field PARMS (in file 9) to isolate the parameters.
- Terminates if no parameter record is found by placing the code E into the first position of the EOF flag field. The first position controls the master file. When the master file reaches the end of file, there are no secondary file coordinations and the program terminates.

Example 4

This example illustrates the use of qualifiers to reference data from multiple files.

```
; Compute total by adding amounts from the file for each quarter.
;
LET T.TOTAL = 1.QTRTOT + 2.QTRTOT + 3.QTRTOT + 4.QTRTOT
```

Note that this example uses the qualifiers to indicate the file from which each quarter's totals are obtained.

The field names within each of the coordinated files are QTRTOT. Each of the files can either use the same definition or different definitions. The qualifier identifies from which of the coordinated files the data is obtained. The temporary field TOTAL uses the qualifier T to identify that it is a computed field. The field is available to all other procedures.

Example 5

This example illustrates the use of the FIND function for performing a file browse.

```
; ***** File Browse Example *****
;
; Report all customers whose customer name begins with 'B'.
; This assumes that the customer name is the primary key of the
; file.
;
IF FIND(1.CUSTOMER WHERE 1.CUSTNAME GE 'B')
   DO UNTIL NOT FIND(1.CUSTOMER)
      IF PF(CUSTNAME 1 1) EQ 'B'
         CALL REPORT LISTCUST
      ELSE
         LEAVE
      END
   END
END
```

In this example, the first statement (IF) reads (FIND) the first record of the file whose key (CUSTNAME) begins with B or greater.

- If no record exists for this condition, the program does not produce report output and terminates the procedures.
- If the first FIND statement reads a record, the customer name (CUSTNAME) is examined to see if it begins with a B.
- If not, the LEAVE statement exits the DO loop and the procedure terminates. Otherwise, the report output is produced and the loop continues by reading (FIND) the next sequential record from the file. The UNTIL NOT FIND clause fails and the loop terminates when the end of the file is reached or the customer name is greater than B.

Note that the program evaluates the UNTIL clause only after the loop executes at least one time.

Example 6

This example is of a GDBI mapping procedure for an ADABAS database.

```
; ***** Root-Level Mapping Procedure for ADABAS *****
;
IF F.COMMAND = 'GETFIRST'
   CALL MODULE INNXXEP T.LOCATE STARTSEG T.RESCODE
   IF T.RESCODE <> 0 OR F.CSTATUS <> ' '
      LET F.MSTATUS = 'STOP10'
      RETURN
   END
END
CALL INNXXEP T.SEQREAD STARTSEG T.RESCODE
IF F.CSTATUS = ' '
   DO CASE
      CASE WHERE T.RESCODE = 0
         T.BCUSED = 0
         RETURN
      CASE WHERE T.RESCODE = -1
         LET F.MSTATUS = 'NFOUND'
         RETURN
   END
END
LET F.MSTATUS = 'STOP20'
```

This procedure retrieves a root segment from the ADABAS database manager and places it into the logical record in field STARTSEG. Based on the flag field COMMAND, the procedure either retrieves the root segment starting from a particular key value or from the beginning of the database.

If the COMMAND flag field contains GETFIRST, the CALL statement interfaces to the INNXXEP module in ADABAS to set the position of the database to the root segment that matches the key value contained in T.LOCATE.

- If the CALL statement is not successful, a value of STOP10 is placed into the MSTATUS flag field and the procedure terminates.
- If the CALL statement is successful or if the COMMAND flag field does not contain GETFIRST, the procedure continues with the second CALL statement that interfaces to the INNXXEP module in ADABAS to actually retrieve the root segment from the current position and return the data in the STARTSEG field. If the second CALL statement is successful, the DO CASE determines if ADABAS found a segment.
  - The programs sets the field T.BCUSED to zero if a segment is retrieved.
  - The program sets the MSTATUS flag field to NFOUND if a segment is not retrieved. In either case, the RETURN statements terminate the procedure.
- If the second CALL statement is not successful, the program places a value of STOP20 into the MSTATUS flag field and the procedure terminates.

Example 7

This example isolates arguments from a text string.

```
;ISOLATE ARGUMENTS FROM A TEXT STRING
;
; INPUT IS T.TEXT, T.START, T.LENGTH
; OUTPUT IS T.ARGST (START OF ARGUMENT)
;           T.ARGL (LENGTH OF ARGUMENT)
;           T.START AND T.LENGTH ARE READY TO CONTINUE THE
;           REMAINDER OF THE SCAN.
; SETTINGS OF ZERO ARE ERROR CONDITIONS
;
LET F.LSTART = T.START                    ;SET SCAN START
LET F.LNUMBER = T.LENGTH                  ;SET SCAN LENGTH
;
IF SCAN(PF(T.TEXT LS LN) FOR P'z')        ;SCAN FOR ALPHA
   LET T.ARGST = F.MSTART                 ;SAVE ARG START
   LET T.LENGTH = F.RNUMBER+F.MNUMBER     ;SET TO SCAN REMAINDER
;                                          OF TEXT
ELSE
   LET T.ARGST = 0                        ;NO ALPHA SO SET ARG
;                                                          PARAMETERS
   LET T.ARGL = 0                         ;TO ZERO
   RETURN
END
;
LET F.LSTART = T.ARGST                    ;SET SCAN START
LET F.LNUMBER = T.LENGTH                  ;SET SCAN LENGTH
IF SCAN(PF(T.TEXT LS LN) FOR P'B')        ;SCAN FOR BLANK
   LET T.ARGL = F.LNUMBER                 ;ARG LENGTH
   LET T.START = F.MSTART                 ;SET NEW START
   LET T.LENGTH = F.RNUMBER + F.MNUMBER   ;SET REMAINDER OF SCAN
ELSE
   LET T.ARGL = T.LENGTH                  ;MUST BE END OF FIELD
   LET T.LENGTH = 0                       ;SET FOR NO MORE TO SCAN
END
```

In this procedure, the SCAN function scans first for an alpha character, as indicated by the pattern P'z' in the first SCAN function. If the program finds an argument, it saves the starting position in the text line; otherwise, the procedure returns and indicates that no argument was found. After an argument is found, the second SCAN function looks for a blank by using the pattern P'B', which indicates the end of the argument.

This routine is used as a subroutine. The calling routine initializes the three temporary fields TEXT, START, and LENGTH. TEXT contains the text to be scanned. START contains the start of the scan. LENGTH contains the length to scan.

With these parameters, the procedure isolates the first word (or token) and returns to the calling routine. When returning to the calling routine, START and LENGTH are adjusted so that, on the next call to this procedure, the scan can continue to find the next word. Thus, successive calls to the procedure return words (or tokens) in sequence.

LENGTH is set to zero when the field TEXT has been completely scanned. The calling procedure tests for this.

Appendix

A Relationship of ASL Statements to Host Engine Statements

This appendix relates the ASL functions and commands to their approximate VISION:Builder statements. They are listed in the order in which they appear in the body of this booklet, grouped by function, then by command.

ASL statements frequently generate more than one VISION:Builder statement. The relationships provided in this appendix are only approximate and are provided for the benefit of those users familiar with fixed format VISION:Builder statements.

Note: Functions and commands pertaining to arrays are applicable to *VISION:Builder* and *VISION:Two*, but not VISION:Inform.

ASL Function	VISION:Builder Operation
FIND	FS when a lower level segment is named. RD, RE, or RG when a root segment from a request read coordinated file is named.
LOOKUP	TL, TN, TS, TB, or TI, depending on the optional keyword used.
PF	A partial field operation is done on the field named in the function. The PF function can be used on as many fields in the command as needed.
SCAN	SL, SR, or SN, depending on the keywords used in the function.
VALIDATE	CV or DV, depending on the keywords used in the function.
CALL	Branches to a subroutine request or procedure, a report, or a subfile. An external program can also be called with this command.
CASE	Creates a logical expression showing what is to be performed when the expression is true.
COLLATE	ER statement type Z.
COMBINE	Cn operator is used with a keyword phrase setting up the number of blanks between fields.
CONTINUE	Performs a GO END procedure.

ASL Command	VISION:Builder Operation
DO	Sets up a loop depending on the keyword phrases used on the command. The keywords UNTIL and WHERE set up back branch loops with the FOR keyword putting a maximum number of back branches to perform. The FORALL keyword sets up an automatic loop for a lower level segment. The CASE keyword sets up a case block that does not establish any loop.
ELSE	Starts the block of code to be performed when the logical expression on the preceding IF command fails or all of the CASE commands within a DO CASE block fail.
END	Delimits the end of a DO block or an IF-THEN-ELSE set of code.
FIELD	TF statement for a temporary field.
GO	Performs a forward branch to a PR statement.
IF	Creates a logical expression showing what is to be performed when the expression is true. Optionally, an ELSE statement can be entered before the corresponding END statement to allow processing of code for a false condition.
LEAVE	Can only be coded in a DO block without the CASE keyword. The DO block is generated as a subroutine, and the LEAVE command in a DO block generates a GO RETURN command.
LET	An R, JR, or JL operation is performed, depending on the keywords at the end of the LET command.
LOCATE	An LR, LC, or LD, depending on the keyword phrases added after the array qualifier is named.
RELEASE	An RS or LA, depending on the keyword phrase provided on the release command.
REPLACE	An SS operator.
RETURN	A GO RETURN from a subroutine procedure.
ROUTE	ER statement type Z.
TRANSFER	A GO NEXT MASTER, GO TYPE 1, or GO TYPE 2.

Appendix

B Technical Notes

This appendix contains additional information about topics described in this book. In some instances, more detail is necessary about the entries that are valid for an operand or more background information might be helpful in understanding a part of the syntax used (for example, qualifiers).

This appendix contains the following topics:

- Reserved Words.
- Qualifiers.
- Patterns.
- Output Edit.
- Valid Field Types and Default Field Lengths.

Reserved Words

Do not make field names the same as system defined names (reserved words) such as commands, functions, keywords, or operators. Making field names the same as system defined names could produce errors.

If you need to, you can use any of these reserved words as a field name if you surround the name by double quotation marks or all keywords are entered in the statement.

The following is a list of restricted ASL words.

ARG
ARGUMENT
ARR
ARRAY
ASTATUS
ATTNID
AVG
BLANKS
CALL
CASE
CHAR
CHKP
CNT
Cn
COL
COLLATE
COLUMN
COM
COMBINE
COMMAND
COND
CONDCODE
CONDITION
CONT
CONTINUE
COUNT
CSTATUS
CUM
CV
DATE
DEC
DECIMALS
DELETE
DLI
DO
DV
ECORD
EDIT
EDLEN
ELSE
END
EOF
EQ
ERROR
EXPR
EXPRESSION
FDNAME
FIELD
FIELDS
FILE
FILEID
FILL
FIND
FIRST
FLD
FLDS
FLOAT
FLT
FOR
FORALL
FROM
FS
GE
GO
GOSC
GOTO
GS
GT
HEAD
HEADING
HEX
HIGHVALU
IF
IN
INIT
INT
INTERPOLATE
ISDATE
JL
JR
JST
JULIAN
JUSTIFY
KEY
LA
LAB
LABEL
LARGER
LAST
LC
LD
LE
LEAVE
LEN
LENGTH
LET
LEVEL
LN
LNUMBER
LOC
LOCATE
LOOKUP
LOWVALU
LR
LRGR
LS
LSTART
LSTATUS
LT
LU
MAX
MIN
MISSPASS
MN
MNUMBER
MOD
MODE
MODULE
MS
MSG
MSGLINE
MSTART
MSTATUS
NE
NEAREST
NEXT
NEXT_PROC
NEXTPROC
NOTEQUAL
NRST
NS
OPERID
OPERL
OUTPUT
OWN

Figure B-1 Reserved Words (Page 1 of 2)

PAGE
PASSWORD
PAT
PATTERN
PCT
PF
PLI
PN
PNUMBER
PROC
PROCEDURE
PS
PSTART
R
RATIO
RD
RE
READ
REL
RELEASE
REP
REPLACE
REPORT
REQ
REQUEST
RESTART
RET
RETURN
RETURNCD
RG
RN
RNUMBER
ROUNDING
ROUTE
ROW
RS
RSTART
RSTATUS
RTO
SCAN
SEG
SEGNAME
SET
SL
SMALLER
SMLR
SN
SR
SS
SSCOUNT
START
STORE
STR
STRAN
STRING
SUB
SUBFILE
TAB
TABLE
TB
TERMID
TEXT
THEN
TI
TIME
TL
TN
TO
TODAY
TOT
TOTAL
TRACE
TRAIL
TRAN
TRANCODE
TRANSFER
TRL
TS
TYPEUNTIL
USING
VAL
VALIDATE
WHERE
WHILE
WITH
XTRAN

Figure B-1 Reserved Words (Page 2 of 2)

Qualifiers

You can use a standard 1-character qualifier and a period to prefix a field name. Qualifiers identify the type of field or file where the field exists. The following are valid qualifiers and their meanings:

VISION:Builder		VISION:Inform	
Qualifier	**Type or Location**	**Qualifier**	**Type or Location**
Blank or N	New master file.	Blank or N	Primary file.
O or 0	Old master file.	O or 0	Primary file.
1-9	Coordinated files 1-9.	1-9	Synchronized files 1-9.
T	Temporary field.	T	Temporary field.
F	Flag field.	F	System field.
X	Transaction file.		
W	Working storage.		
V	Linkage section.		
A, B, E, G, H, J, K, M, Q, 1-9	Array (must match the qualifier that identifies the array).		

Patterns

There are two pattern types for VISION:Builder and VISION:Inform applications: validation patterns and edit patterns. This section discusses each pattern type.

Validation Patterns

Use validation patterns in SCAN, VALIDATE, and REPLACE statements. Enclose patterns in single quotation marks, preceded by a P. You can make patterns up to 30 characters long. Code one pattern symbol for each character of the field being validated.

Note:

- A minus (–) sign before any of these characters means scanning for other than the specified pattern. Minus C (-C) is not a legal entry.
- You must use system delimiters to surround literals in a scan pattern.

You can use the following validation pattern symbols:

Symbol	Meaning
Z	Alpha (A-Z).
z	Alpha (A-Z, a-z).
A	Alpha (A-Z) or blank.
a	Alpha (A-Z, a-z) or blank.
D	System delimiter.
9	Numeric (0-9).
I	Numeric (0-9) or blank.
Y	Alpha (A-Z) or numeric (0-9).
y	Alpha (A-Z, a-z) or numeric (0-9).
X	Alpha (A-Z), numeric (0-9), or blank.
x	Alpha (A-Z, a-z), numeric (0-9), or blank.
B	Blank.
C	No validation.
Literal(s)	Any character(s).
User-defined	Provided by you at system installation time.

Edit Patterns

Create edit patterns to edit the result field in the LET statement. Using edit patterns, you can specify whether to store additional characters into the result field or truncate existing ones. You can make the edit pattern up to 30 characters long.

There are two styles of patterns available: the delimiter style or the COBOL style.

- Delimiter style patterns use the delimiter character as the character/digit selector symbol, the decimal point as a decimal alignment symbol, and all other characters as literal insertion characters. Use this edit pattern style when editing character data or when editing numeric data.
- COBOL style patterns use the symbols described in Numeric Data (Packed, Zoned, and Fixed Point Binary Only). Use this edit pattern style only with numeric data.

Character String Data

Enter the edit pattern as a picture, with the delimiter (#) as an edit pattern symbol. If required, you can insert additional edit symbols. For example, you can specify a pattern as ##/##/##.

- The first character in the edit pattern represents the character on the extreme left.
- Truncate characters on the right by not entering delimiters to represent them.
- If left truncation or extraction of characters in the middle of a field is preferred, use partial fielding to specify the partial field start position and number of characters to be stored.

Numeric Data (Packed, Zoned, and Fixed Point Binary Only)

Use the edit pattern to truncate digits following the decimal point or non-significant zeroes to the left of the decimal point. The pattern causes digits surrounding the decimal point to be stored.

The period (.) has special meaning in the edit pattern.

- The first period is interpreted as a printable decimal point and is used to align decimal points.
- If no period exists, only the integer portion of a number is stored.

Leading zeroes are suppressed if they are the leading characters of the picture. When a negative value is output, the sign is stored to the left of the first digit or replaces the first leading zero.

You can use the following edit pattern symbols for numeric data:

Symbol	Meaning
9	Digit select. Represents a character position that contains a numeral.
Z	Digit select. Same as 9, except when the result contains a zero as a leading character, the zero is replaced by a space (blank). Signs do not print.
*	Check protection. Same as Z, except a zero in a leading position is replaced by an asterisk (*).
.	Decimal position. Represents a decimal point for alignment in the edit picture. For the output, the decimal point represents a position into which a character is inserted.
,	Grouping character. Represents an output position into which a grouping character is placed.
+ –	Sign control. These symbols are used as editing sign control symbols to position the edit sign.
$	Currency symbol. Represents an output position into which the currency symbol is placed.

Rules for Edit Patterns

- You can edit source digits to the output positions with or without suppression of leading zeroes or check protection (for example, 99.9 = no suppression, ZZ.9 = suppress leading zeroes to decimal point, **.* = check protects the entire field).
- You can specify a leading or trailing sign. For example, +99, –99, 99–, 99+.

Symbol in Edit Picture	Result if Positive Data Item (Includes Zero)	Result if Negative Data Item
+	+	–
–	SPACE	–

- You can specify a leading sign as fixed (leftmost position on output) or floating. For example, +99.9 = fixed, +++.9 = floating.
- You can specify a currency symbol as fixed (leftmost position on output) or as floating (for example, $99.9 = fixed, $$$.9 = floating). The currency symbol and the sign symbols are mutually exclusive as fixed insertion characters in a given picture.
- Indicate floating insertion of leading sign or currency symbol by a string of at least two occurrences of the same characters ($+–) to represent the leftmost numeric position in the output. For example, +99.9 = fixed, +++.9 = floating.
  - The currency symbol and the sign symbols are mutually exclusive as floating insertion characters in a given picture.
  - A fixed insertion character can also be specified in the same picture with a floating insertion character as long as it is a different character.
- Use floating insertion of leading sign or currency symbols to suppresses leading zeroes in the same manner as the digit select character Z.
- Suppression symbols (Z$*+–) can occur in an edit picture as follows:
  - Any or all of the leading numeric positions to the left of the decimal point are represented by suppression symbols (for example, $$,$$$.99).
  - All of the numeric positions in the edit picture are represented by suppression symbols (for example, $$,$$$.$$).

  If the suppression symbols appear to the left of the decimal point, any leading zero in the data that appears in a character position corresponding to a suppression symbol in the picture is replaced by the space character or check protection character.

Suppression terminates at the first non-zero digit in the data or at the decimal point, whichever comes first. For example, $$$.99 with data 001.23 produces $1.23.

- If all numeric positions in the picture are represented by suppression symbols and the value of the data is not zero, the result is the same as if the suppression characters were to the left of the decimal point.
- If the value of the data is zero, the entire output becomes spaces if all numeric positions in the picture are represented by suppression symbols. For example, $.$$ with data 0.00 produces all spaces.
- If check protection is used, all positions become the protection character, except for a decimal character. For example, *.** with 0.00 produces *.**.

Any grouping characters embedded in the string of suppression characters or to the immediate right of the string are suppressed if the left adjacent numeric position is suppressed. For example, $,$$$.99+ with data 0012.34 produces $12.34+.

- Using a suppression symbol throughout a field suppresses a trailing sign on all zero fields (for example, ZZ.ZZ+ with value +00.01 produces .01+, while +00.00 produces all spaces). Delimiters should not be used with Z picture edit as ZZZ,ZZZ.##, but rather, as ZZZ,ZZZ.99.
- The grouping character can occur on either side of the decimal position character (for example, $$,$$$.999,99 international metric standard).
- Truncating the significant trailing digits is valid (for example, 99.99 with data 12.345 produces 12.34). Truncating the significant leading digits is not valid. For example, 9.99 with data 12.34 produces + or the “uneditable" VISION:Builder symbol.
- A trailing character, if specified, cannot be the same symbol used as a floating or leading character in a given pattern. For example, +99.9+ is invalid; +99.9– is valid.

Output Edit

Use the output edit entries (floating, filling, or trailing) only with numeric fields (packed, zoned, and fixed point binary). These entries edit the data before reporting it.

Commas

Commas print in these fields where they are preceded by a significant digit. To suppress printing of commas, use edit suppress character 'Z' or the Override "Picture" Edit on the reporting statement.

Standard Notation

Do not make an entry for floating point numbers. They always print in scientific notation. The output edit length must exceed six to include two signs, a decimal point, E, and two exponent digits. Standard notation for floating point numbers, where Xs represent the fraction and Ys the exponent, is:

± .XXXXXXXE ±YY

Floating/Edit Suppress

An entry other than Z "floats" the value of the entry and prints to the immediate left of the first significant digit in the printed report.

- If this is not used, a leading blank prints if positive, a minus if negative.
- If this operand contains a Z, all commas, leading zeroes to the left of the decimal point, and the decimal point, if specified, are suppressed.

Float (Floating-Edit-Char)

This operand is valid only for packed, zoned, and fixed point binary field types.

Code	Result
Blank	When no output codes are specified, a leading blank prints if the value of the field is positive; a leading minus sign prints if the value of the field is negative; commas print and the decimal point prints if the decimal places are specified. A zero value, in a field where a decimal place is specified, prints as a decimal point followed by as many zeroes as there are decimal places.
$	A floating dollar sign prints before the first value when a control break occurs and when summaries are taken.
+	A leading + sign prints if the value of the field is positive, – if negative.

Code	Result
–	A leading – sign prints if the value of the field is negative (default specification).
Z	The printing of leading zeroes is suppressed from the left to the first non-zero digit or decimal place. Negative signs print, but no space is allocated for them.
Any other character	A floating leading character prints on control breaks and on summaries.

Fill (Fill-Edit-Char)

The value of this operand prints in every position from the leftmost portion of the field until the first non-zero digit is encountered. This operand is valid only for packed, zoned, and fixed point binary fields.

Code	Result
Any character	Replaces leading zeroes.

Trail (Trailing-Edit-Char)

The value of this operand prints following positive and/or negative values. This operand is valid only for packed, zoned, and fixed point binary fields.

Code	Result
+	A trailing + sign prints if the value of the field is positive.
–	A trailing – sign prints if the value of the field is negative.
)	Encloses negative field values in parentheses. If no filling character is specified, the left parenthesis prints before the first significant digit or decimal point, whichever comes first. If a floating character and this character are specified, both can print. The floating character prints inside the parentheses: ($43.50). Only a single floating sign is permissible with the trailing “)".
C	Prints a trailing “CR" for a negative value. Blanks follow a positive value.
D	Prints a trailing “DB" for a negative value. Blanks follow a positive value.
Any other character	Prints a trailing character for negative values only.

Edlen (Edit-Length)

The number of print positions for reporting the field.

Code	Result
Blank	For normal or fixed length fields, the system computes the length. For variable length fields, the system computes the length as the shorter of either the defined field length or report page width minus spaces before columns.
Any two-digit number	For normal or fixed length fields, the system uses the output edit length or field, including any floating, filling, and/or trailing characters. For variable length fields, the system uses the width of the column. Text automatically wraps until it is exhausted.

Valid Field Types and Default Field Lengths

The following table gives the valid entries for field type and field length and the defaults for field length.

FIELD TYPE	FIELD RANGE	DEFAULT FIELD LENGTH
C Character	1-255 Bytes	16
Z Zoned	1-15 Bytes	15
P Packed	1-15 Bytes	8
V Variable	1-999 or 1H-99H (H implies 00)	NULL
E Floating point binary	4-Bytes only	4
F Fixed point binary	1-4 Bytes	4

FIELD TYPE	FIELD RANGE	DEFAULT FIELD LENGTH
L Time HOUR:MIN:SEC	$\left[\frac{Nn + 7}{2}\right]$ to 8	8
	$\left[\frac{Nn + 5}{2}\right]$ to 8	8
S Time MIN:SEC	Where Nn is the number of decimal-of-seconds digits. (Nn ≤ 9) and [ ] around the equation means use only the integer portion of the number.	

Appendix

C Flags

A flag is an internal indicator defined by VISION:Builder, VISION:Inform, and VISION:Two, which designates the existence of certain conditions during the application run. Special words function as flags. The names of the flags are not reserved; you can use flags as field names without restriction. You can also use flags in processing and reporting statements. The F qualifier identifies a flag to VISION:Builder.

Note:

- Flags pertaining to arrays are not applicable to VISION:Inform.
- In this appendix, the symbol ❹ or (*VISION:Builder* 4000 Model Series Only) designates that the feature or function being described only applies to the *VISION:Builder* 4000 model series and not to the *VISION:Two* 2000 model series.

The VISION:Builder software system has the following processing options, available in both the 4000 and 2000 model series.

- IMS™ — The IMS option provides support for processing information in IBM IMS Databases using the standard DL/I processing facilities.

  In this appendix, the symbol ❶ or (IMS Only) designates that the specification, feature, or function being described only applies to the IMS option.

- DB2 — The DB2 option provides support for processing information in IBM Database2 Relational Tables using the standard SQL processing facilities.

  In this appendix, the symbol ❷ or (DB2 Only) designates that the specification, feature, or function being described only applies to the DB2 option.

- GDBI — The GDBI option provides a facility for interfacing user code with the standard mechanisms of VISION:Builder to perform I/O operations using any database processing facilities.

  In this appendix, the symbol ❻ or (GDBI Only) designates that the specification, feature, or function being described only applies to the GDBI option.

Flag	Application	Format
ASTATUS	Indicates the status of an array operation.	4 bytes character
CHKP[1]	Directs VISION:Builder to take a checkpoint before reading the next master file root segment.	1 byte character
CKPTID[1]	Contains the checkpoint ID of the last checkpoint taken.	8 bytes character
COLUMN	Indicates the number of the current column of the array.	4 bytes fixed
COMMAND[G]	Contains the command identifier.	8 bytes character
CONDCODE	Modifies VISION:Builder condition codes (OS/390, CMS), terminates job at end of job step (VSE). (User modifiable)	2 bytes binary
CSTATUS	Indicates that a CALL was suppressed by VISION:Builder and gives the reason.	4 bytes character
DATE	Records the operating system date in the format: MMM DD, YYYY.	12 bytes character
DELETE[4]	Directs VISION:Builder to delete the current master file record, to delete all segment occurrences having empty key field values, or to reject the current transaction.	1 byte fixed
ECORD	Indicates the match condition between the master record key field and the key fields of the current coordinated records.	9 bytes character
EOF	Detects or forces end of file on any sequentially input files. (User modifiable)	11 bytes character
FDNAME[G]	Contains the file name of the mapped file.	8 bytes character
FILE[G]	Contains the VISION:Builder logical file name.	8 bytes character
FILEID[G]	Contains the file identification of the mapped file.	8 bytes character

Flag	Application	Format
ISDATE	Records the operating system date in the format:	
	■ YYYYMMDD (normal usage)	8 bytes character
	■ YYYY-MM-DD (formatted reports)	10 bytes character
JULANX	Records the operating system date in the format:	
	■ YYYYDDD (normal usage)	7 bytes character
	■ YYYY.DDD (formatted reports)	8 bytes character
JULIAN	Records the operating system date in the format:	
	■ YYDDD (normal usage)	5 bytes character
	■ YY.DDD (formatted reports)	6 bytes character
LILIAN	Date type field containing the Lilian date as the number of days since the beginning of the Gregorian calendar (October 14, 1582). The valid range of Lilian dates is 1 – 3,074,324 (October 15, 1582 to December 31, 9999). For example, a Lilian date with a value of 152384 converts to the standard date of December 31, 1999.	4 bytes integer (date type)
LNUMBER	Indicates the number of characters in the left part of the field just scanned. (User modifiable)	4 bytes fixed
LSTART	Indicates the starting location of the left part of the field just scanned. (User modifiable)	4 bytes fixed
LSTATUS	Indicates the status of segment operations.	4 bytes character
M4AUDIT ❹	Provides the number of deleted master file records output to M4AUDIT during application execution.	4 bytes fixed
M4CORD1	Provides the number of records read from M4CORD1 during application execution.	4 bytes fixed
M4CORD2	Provides the number of records read from M4CORD2 during application execution.	4 bytes fixed
M4CORD3	Provides the number of records read from M4CORD3 during application execution.	4 bytes fixed

Flag	Application	Format
M4CORD4	Provides the number of records read from M4CORD4 during application execution.	4 bytes fixed
M4CORD5	Provides the number of records read from M4CORD5 during application execution.	4 bytes fixed
M4CORD6	Provides the number of records read from M4CORD6 during application execution.	4 bytes fixed
M4CORD7	Provides the number of records read from M4CORD7 during application execution.	4 bytes fixed
M4CORD8	Provides the number of records read from M4CORD8 during application execution.	4 bytes fixed
M4CORD9	Provides the number of records read from M4CORD9 during application execution.	4 bytes fixed
M4NEW ❹	Provides the number of records output to M4NEW during application execution.	4 bytes fixed
M4OLD	Provides the number of records read from M4OLD during application execution.	4 bytes fixed
M4REJECT ❹	Provides the number of records output to M4REJCT during application execution.	4 bytes fixed
M4SUBF0	Provides the number of records output to M4SUBF0 during application execution.	4 bytes fixed
M4SUBF1	Provides the number of records output to M4SUBF1 during application execution.	4 bytes fixed
M4SUBF2	Provides the number of records output to M4SUBF2 during application execution.	4 bytes fixed
M4SUBF3	Provides the number of records output to M4SUBF3 during application execution.	4 bytes fixed
M4SUBF4	Provides the number of records output to M4SUBF4 during application execution.	4 bytes fixed
M4SUBF5	Provides the number of records output to M4SUBF5 during application execution.	4 bytes fixed
M4SUBF6	Provides the number of records output to M4SUBF6 during application execution.	4 bytes fixed
M4SUBF7	Provides the number of records output to M4SUBF7 during application execution.	4 bytes fixed
M4SUBF8	Provides the number of records output to M4SUBF8 during application execution.	4 bytes fixed

Flag	Application	Format
M4SUBF9	Provides the number of records output to M4SUBF9 during application execution.	4 bytes fixed
M4TRAN ❹	Provides the number of records read from M4TRAN during application execution.	4 bytes fixed
MISSPASS	Indicates the status of the follow-up pass in sequential coordination for master file records.	1 byte character
MNUMBER	Indicates the number of characters in the middle part of a field just scanned. (User modifiable)	4 bytes fixed
MODE Ⓖ	Contains information about the application mode of operation.	2 bytes character
MSTART	Indicates the starting location of the middle part of a field just scanned. (User modifiable)	4 bytes fixed
MSTATUS Ⓖ	Used by the mapping request to instruct VISION:Builder on a specific action to take subsequent to completion of the mapping request. (User modifiable)	6 bytes character
OWN	Provides a means of communication between user routines and own-code exits. (User modifiable)	16 bytes character
PAGE	Indicates placement of page numbers in formatted reporting.	6 bytes character
PASSWORD Ⓖ	Contains the password from the RF statement.	8 bytes character
RESTART ❶	Indicates restart ID or blank if not a restart.	8 bytes character
RETURNCD	Determines the results of the CALL as set by the generalized system interface CALL routine.	4 bytes fixed
RNUMBER	Indicates the number of characters in the right part of a field just scanned. (User modifiable)	4 bytes fixed
ROW	Indicates the row number in the array.	4 bytes fixed
RSTART	Indicates the starting location of the right part of the field just scanned. (User modifiable)	4 bytes fixed
RSTATUS	Indicates the results of a read operation.	4 bytes character
SEGNAME Ⓖ	Contains the segment name as defined on the LS statement.	8 bytes character

Flag	Application	Format
SQL ❹	Indicates the status of inserting a row into an SQL table.	4 bytes character
SSCOUNT	Counts the number of matches found during a REPLACE operation.	2 bytes fixed
STRAN ❹	Interrogates status of transactions at each segment and indicates which have been applied to the master file.	9 bytes character
TIME	Records the time of day a job was started in the format: HH.MM.SS (hours, minutes, seconds).	8 bytes character
TODAY	Records the operating system date in the format:	
	■ MMDDYY (normal usage)	6 bytes character
	■ MM/DD/YY (formatted reports)	8 bytes character
TODAYX	Records the operating system date in the format:	
	■ MMDDYYYY (normal usage)	8 bytes character
	■ MM/DD/YYYY (formatted reports)	10 bytes character
TRAN ❹	Indicates the status of a master file record and/or the rejection of a transaction.	1 byte fixed
XTRAN ❹	Identifies the reason for rejection of a particular transaction.	1 byte fixed

ASTATUS Flag

The ASTATUS flag indicates the status code following the execution of an array operation. It is a character field with a length of 4 bytes. The status information can be examined to determine if the operation was successful, failed, or was suppressed.

- Use the ASTATUS flag to debug applications or to determine sources of erroneous input to applications.
- As part of the array operation, there is an implied compare to blanks in the ASTATUS flag when there is an NS or GS operation immediately following an array operation. The NS or GS branch will be taken on any status other than blanks. At the "branch-to" location, the status information can be examined to determine why the operation failed or was suppressed.

The ASTATUS codes and explanations are shown in the following table.

	Operations Involved				
Value	Locate Row and Column	Locate Row	Locate Column	Release	Comments
␣␣␣␣	Yes	Yes	Yes	Yes	The operation executed successfully.
BMIS	Yes	Yes	No	No	The operation is unsuccessful. The value of the indicated row or column is missing or invalid as indicated or is larger than a 4-byte fixed point field. If the first character is B, the error applies to the row. If it is C, the respective error applies to the column.
BINV	Yes	Yes	No	No	
CMIS	Yes	No	Yes	No	
CINV	Yes	No	Yes	No	

Value	Operations Involved: Locate Row and Column	Locate Row	Locate Column	Release	Comments
ROWH ROWL COLH COLL	Yes Yes Yes Yes	Yes Yes No No	No No Yes Yes	No No No No	The operation is unsuccessful. The value supplied for row or column is not within the dimensions of the array. The value is greater than the highest row number. The row value is less than 1. The value is greater than the highest column number. The column value is less than 1.
AUTO	Yes	Yes	Yes	Yes	The operation is suppressed. An automatic loop is active on the array.
INHR	Yes	Yes	Yes	Yes	The operation is suppressed. A subroutine was called; it attempted to perform an array operation on an array that was already positioned to a specific data cell due to field references in the calling request, or a subset of the array was located by a calling request by means of a successful array operation.

CHKP Flag (IMS Only)

The CHKP flag triggers a checkpoint operation by placing a non-blank value into the flag from any request except preselection (type P). It is a character field with a length of 1 byte. The checkpoint does not occur until just before VISION:Builder is ready to read the next master file root segment. The flag is reset to a blank after every checkpoint.

The user can place an "A" in the CHKP flag to force an ABEND to occur instead of a checkpoint. The ABEND code issued will be 117.

CKPTID Flag (IMS Only)

The CKPTID flag contains the ID of the last checkpoint taken. It is a character field with a length of 8 bytes. It contains blanks prior to the first checkpoint.

COLUMN Flag

The COLUMN flag is set after the successful completion of an array operation. It is a fixed field with a length of 4 bytes. It contains a numeric value set to the column number being processed.

- If the operation fails, the flag is set to zero.
- If an array operation is suppressed, the COLUMN flag value is not changed.

COMMAND Flag (GDBI Only)

The COMMAND flag G contains the command identifier. It is a character field with a length of 8 bytes. Flag values are:

Input:	Sequential or Serial:	GETFIRST	Get first segment within a parent.
		GETNEXT	Get the next occurrence of the segment type previously obtained.
	Key driven: (transaction driven, 4 ICF, start search)	GETFKEY	Get first segment with key greater than or equal to the supplied value.
		GETKEY	Get the segment with the key provided.
Output:		REPLACE 4	Replace the segment. Some data has been changed.
		ADD 4	Add the segment. A new segment has been created.
		NOCHANGE 4	This segment has not changed (however, BDAM may still need to see it).
		DELETE 4	Delete the segment. This segment was explicitly deleted by the application by a transaction or DELETE flag setting. The parent segment has not been deleted.
		INIT	Initialization request call.
		TERM	Termination request call.

The COMMAND flag settings are not strictly necessary if LM statements are used. Some mapping developers may prefer to write a single request to handle all situations. In this case, the flag becomes valuable to identify the activity required.

The information is primarily useful for directing the operation of the mapping request, but may also be useful to a database manager. The flags are initialized prior to entering a mapping request.

CONDCODE Flag

Use the CONDCODE flag to communicate with your job control. It is a binary field with a length of 2 bytes. The flag has an external effect only on OS/390, CMS, and VSE.

Note: It is your responsibility to ensure that the final value of the condition code is recognized by the operating system.

You can access this flag during processing for arithmetic calculations, selection, and output. It is initialized to zero at the beginning of the run. During processing, it contains only the values supplied by you. The values normally supplied by VISION:Builder are not available during request processing.

VSE: Placing any non-zero numeric value in the CONDCODE flag cancels the job due to program control at the end of the current step. Thus, you can cancel a job by replacing 1 into CONDCODE and setting the EOF flag to all Es.

OS/390 and CMS: At the end of the run, after all request processing is completed and all files have reached end of file, the values you placed in the CONDCODE flag during request processing are added to the condition code values normally supplied by VISION:Builder.

- Use this procedure to control the range of condition codes without losing the settings provided by VISION:Builder.
- Only the last value of CONDCODE is used; intermediate values have no effect.
- If the CONDCODE flag is invalid at the end of the run,VISION:Builder sets it to 20 before adding it to the normal condition code value. In CMS, the condition code is referred to as the return code.

CSTATUS Flag

The CALL status flag (CSTATUS flag) indicates the status of a CALL. It is a character field with a length of 4 bytes. The CSTATUS flag values are set by VISION:Builder and indicate whether a CALL was suppressed by VISION:Builder and why.

- A CALL is suppressed if a parameter specifies an invalid or missing field. The following table shows the settings of the CSTATUS flag:

Value	Meaning
PMIS	Parameter missing.
PINV	Parameter invalid.
blank	CALL successful.

- The CSTATUS flag should be examined before using a value from the RETURNCD flag, because a suppressed call would not allow the called routine to set the RETURNCD flag.

DATE Flag

The DATE flag records the date, acquired from the operating system, in the format: MMM DD, YYYY, where MMM equals 3 alpha characters for month, DD equals 2 numerals for day of month, and YYYY equals 4 numerals for year (that is, January 15, 2001 equals JAN 15, 2001). It is a character field with a length of 12 bytes.

DELETE Flag (VISION:Builder 4000 Model Series Only)

The DELETE flag ❹ directs VISION:Builder to delete records or segment occurrences from a master file or to force rejection of a transaction record during processing. It is a fixed field with a length of 1 byte. The settings for the DELETE flag are shown in the following table.

Setting	Result
0	All M4OLD data is output to M4NEW. The DELETE flag is automatically set to zero each time a record is processed.
1	The master file record is deleted before it can be written to the new master file following standard request processing. Records can also be dynamically deleted from the master file in type M, 2, and 3 procedures/requests. Deleted records can be output to an audit file.
2	The appropriately marked lower level segments are deleted from the new master record before it is written to the new master file. Segments are marked for deletion procedurally by placing blank or zero into the segment key and placing 2 into the DELETE flag. Segments can be marked for deletion in type N, M, 2, and 3 procedures/requests. This is not supported when processing a relational file.
4	The transaction is rejected without the master file being updated. This occurs only in type 1 and 2 procedures/requests.

ECORD Flag

The ECORD flag indicates the status of the coordinated files in relation to the master file or coordinated file to which it is chained. It is a character field with a length of 9 bytes. The values for the ECORD flag are shown in the following table.

Value	Meaning
M	The coordinated file and master file keys are equal. The coordinated file has a match and its record is available for processing.
X	The coordinated file is high in relation to the master file key. The coordinated file record is not available for processing. This value is also used when the coordinated file reaches the end of file, when no coordinated file exists for the run, or if the file is request read.
L	The coordinating file key is less than the master file key. The record is available for processing.

In the ECORD flag, each character indicates the current status of a corresponding coordinated file. Partial fielding is used to determine the status of one or more of the coordinated files, as shown below.

File Name	Partial Field Start and Length
Coordinated File 1	1,1
Coordinated File 2	2,1
Coordinated File 3	3,1
Coordinated File 4	4,1
Coordinated File 5	5,1
Coordinated File 6	6,1
Coordinated File 7	7,1
Coordinated File 8	8,1
Coordinated File 9	9,1

The ECORD flag is initialized whenever a master file record is read or created. For each coordinated file specified on an RF statement, the ECORD flag is initialized to X. As each coordinated file record is advanced, its status (in relation to the master file record being matched) is reflected in the ECORD flag.

The ECORD flag indicates file status and must not be altered. Altering the ECORD flag does not change the physical status of the files, but misleads subsequent processing.

EOF Flag

Use the EOF flag to detect or force an end of file on sequentially read input files. It is a character field with a length of 11 bytes. You can test this flag to determine if an input file exists. Alternatively, you can use it to terminate the reading of an input file by storing the appropriate value in the flag.

Note: When forcing end of file, if any value except E is placed into the EOF flag, the results are unpredictable.

Each character pertains to an input file as shown in the following table and can have one of three values:

Value	Meaning
E	File has reached end of file.
Y	File has not reached end of file.
N	File does not exist.

Input File Name	Partial Field Specification
M4OLD	1,1
M4TRAN	2,1
M4CORD1	3,1
M4CORD2	4,1
M4CORD3	5,1
M4CORD4	6,1
M4CORD5	7,1
M4CORD6	8,1
M4CORD7	9,1
M4CORD8	10,1
M4CORD9	11,1

FDNAME Flag (GDBI Only)

The FDNAME flag G contains the file name as in columns [1-8] on the FD statement. It is a character field with a length of 8 bytes.

It is initialized prior to entering a mapping request. The information in the flag is generally required by database managers.

FILE Flag (GDBI Only)

The FILE flag G contains the VISION:Builder logical file name. It is a character field with a length of 8 bytes. Values are: M4OLD, M4NEW, and M4CORD1–9.

The information is primarily useful for directing the operation of the mapping request, but can also be useful to a database manager.

FILEID Flag (GDBI Only)

The FILEID flag G contains the file identification as provided in columns [11-18] of the FD statement. It is a character field with a length of 8 bytes.

It is initialized prior to entering a mapping request. The information in the flag is generally required by database managers.

ISDATE Flag

The ISDATE flag records the system date in two International Standards Organization (ISO) formats.

Note: The date delimiter (-) is an installation option and can be changed using M4PARAMS. See the *Installation Guide*.

Normal processing/standard reporting (8 characters)

This flag records the date in the format of YYYYMMDD, where YYYY equals 4 numerals for year, MM equals 2 numerals for month, and DD equals 2 numerals for day (that is, January 15, 2001 equals 20010115).

Formatted reporting (10 characters)

This flag records the date in the format of YYYY-MM-DD (2001-01-15).

JULANX Flag

The JULANX flag records the system date in two formats.

Note: The date delimiter (.) is an installation option and can be changed using M4PARAMS. See the *Installation Guide*.

Normal processing/reporting (7 characters)

This flag records the date in the format of YYYYDDD, where YYYY equals year and DDD equals day in the year (that is, January 15, 2001 equals 2001015).

Formatted reporting (8 characters)

This flag records the date in the format of YYYY.DDD (2001.015).

JULIAN Flag

The JULIAN flag records the system date in two formats.

Note: The date delimiter (.) is an installation option and can be changed using M4PARAMS. See the *Installation Guide*.

Normal processing/reporting (5 characters)

This flag records the date in the format of YYDDD, where YY equals year and DDD equals day in the year (that is, January 15, 2001 equals 01015).

Formatted reporting (6 characters)

This flag records the date in the format of YY.DDD (01.015).

LILIAN Flag

The LILIAN flag contains the current date in Lilian date format as a fixed-binary number with a length of 4 bytes. The valid range of Lilian dates is 1 – 3,074,324 (October 15, 1582 to December 31, 9999). VISION:Builder automatically converts a LILIAN flag field into character format using M4PARAMS settings when a LILIAN flag field is specified for printing. The format is identical to the TODAYX flag conventions unless an override edit picture is specified.

LNUMBER FLAG

The LNUMBER flag is a text processing partial field flag that specifies the number of characters in the left part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value LN can be used in place of the standard partial field specification in order to access a particular area of the scanned field.
- The value LN (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if LN is entered in starting character on the PR statement, the current value of LNUMBER is used at the time the PR statement is executed.

Because the value of LNUMBER may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation, where the result field is invalid for the current operation only).

LSTART Flag

The LSTART flag is a text processing partial field flag that specifies the starting position of the left part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value LS can be used in place of the standard partial field specification on the processing and record selection statement in order to access a particular area of the scanned field.
- The value LS (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if LS is entered in starting character on the PR statement, the current value of LSTART is used at the time the PR statement is executed.

Because the value of LSTART may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation where the result field is invalid for the current operation only).

LSTATUS Flag

The LSTATUS flag indicates the status code following the execution of any segment operation. It is a character field with a length of 4 bytes. The status information can be examined to determine if the operation was successful, failed, or suppressed. This flag and its contents are independent of the type of file being accessed (for example, IMS, sequential, ISAM).

As part of the segment operation, there is an implied compare to blanks in the LSTATUS flag when there is an NS or GS operation immediately following a segment operation. The NS or GS branch will be taken on any status other than blanks. At the "branch-to" location, the status information can be examined to determine why the operation failed or was suppressed. You can use this flag in debugging applications or in determining sources of erroneous input to applications.

The LSTATUS codes and explanations are shown in the following table. The priority column indicates the order in which the LSTATUS flag settings occur.

LSTATUS Value	Applicable To FS	FF FL	RS	Priority	Explanation
b/b/b/b/	Yes	Yes	Yes		The operation executed successfully.
NREC	Yes	Yes	No	1	The operation is suppressed. The current record is unavailable for the referenced segment.
AUTO	Yes	Yes	Yes	2	The operation is suppressed. The segment type referenced is already in an automatic loop; or a field in operand B is part of the segment type referenced, or part of an unreferenced (by an FS or RS operator) dependent segment.
TRAN ❹	Yes	Yes	Yes	3	The operation is suppressed. The segment type referenced was matched, created, or inserted by the current transaction record. It can only be set in the type M procedures or its subroutines.
TDEL ❹	Yes	Yes	Yes	3	The operation is suppressed. The segment type referenced was deleted by the current transaction record. It can only be set in the type M procedure or its subroutines.
INHR	Yes	Yes	Yes	4	The operation is suppressed. The segment type referenced in a subroutine procedure is in an automatic loop inherited from the calling procedure.
BMIS	Yes	No	No	6	A field in the operand B has a missing value.
BINV	Yes	No	No	6	A field in the operand B has an invalid value.
NPAR	Yes	Yes	Yes	7	The operation is suppressed. The segment type referenced has a parent segment that is unavailable or does not exist.
NFND	Yes	Yes	No	8	The operation fails. VISION:Builder cannot find any segment occurrences that satisfy the operation.
NOMO	Yes	No	No	8	The operation fails. The next occurrence of a referenced segment type does not exist.

M4AUDIT Flag (VISION:Builder 4000 Model Series Only)

The M4AUDIT flag ❹ provides access to the number of records output to the M4AUDIT file during application execution. It is a fixed field with a length of 4 bytes. The M4AUDIT flag can be referenced in any type of request (except preselection); however, it can only be referenced if the M4AUDIT file is being used in the application.

M4CORDn (n=1 to 9) Flags

There are separate flags provided for each one of the nine VISION:Builder coordinated files, M4CORDn where n=1 to 9. These are fixed fields with lengths of 4 bytes.

These flags (M4CORD1, M4CORD2, M4CORD3, M4CORD4, M4CORD5, M4CORD6, M4CORD7, M4CORD8, and M4CORD9) provide access to the number of records read from each of the various coordinated files during application execution. These coordinated file flags can be referenced in any type of request (except preselection); however, they can only be referenced if the coordinated file they reference is being used in the application.

M4NEW Flag (VISION:Builder 4000 Model Series Only)

The M4NEW flag ❹ provides access to the number of records output to the M4NEW file during application execution. It is a fixed field with a length of 4 bytes. The M4NEW flag can be referenced in any type of request (except preselection); however, it can only be referenced if the M4NEW file is being used in the application.

M4OLD Flag

The M4OLD flag provides access to the number of records read from the M4OLD file during application execution. It is a fixed field with a length of 4 bytes. The M4OLD flag can be referenced in any type of request (except preselection); however, the flag can only be referenced if the M4OLD file is being used in the application. When using update-in-place, the flag will represent only the records read from M4OLD.

M4REJECT Flag (VISION:Builder 4000 Model Series Only)

The M4REJECT flag❹ provides access to the number of records output to the M4REJCT file during application execution. It is a fixed field with a length of 4 bytes. The M4REJECT flag can be referenced in any type of request (except preselection); however, it can only be referenced if the M4REJCT file is being used in the application.

M4SUBFn (n= 0 to 9) Flags

There are separate flags provided for each one of the ten VISION:Builder subfiles, M4SUBFn where n=0 to 9. These are fixed fields with lengths of 4 bytes.

These flags (M4SUBF0, M4SUBF1, M4SUBF2, M4SUBF3, M4SUBF4, M4SUBF5, M4SUBF6, M4SUBF7, M4SUBF8, and M4SUBF9) provide access to the number of records output to each of the various subfiles during application execution. These subfile flags can be referenced in any type of request (except preselection); however, they can only be referenced if the subfile they reference is being used in the application.

M4TRAN Flag (VISION:Builder 4000 Model Series Only)

The M4TRAN flag❹ provides access to the number of records read from the M4TRAN file during application execution. It is a fixed field with a length of 4 bytes. The M4TRAN flag can be referenced in any type of request (except preselection); however, it can only be referenced if the M4TRAN file is being used in the application.

MISSPASS Flag

The MISSPASS flag value is automatically set by VISION:Builder only at the start of the all-miss pass (ECORD setting of all Xs). It is a character field with a length of 1 byte. This flag will contain either an X or an M during the all-miss pass. An X setting indicates that no hits occurred during coordination for the master record; an M setting indicates at least one hit occurred during coordination. The MISSPASS flag is blank when the all-miss pass is not in progress.

While the ECORD flag indicates the status of coordinated files, the MISSPASS flag indicates the status of the master file. The MISSPASS flag should not be used with RPCORDONLY, because its value during a match or low cycle is blank and meaningless.

MNUMBER Flag

The MNUMBER flag is a text processing partial field flag that specifies the number of characters in the middle (matching) part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value MN can be used in place of the standard partial field specification in order to access a particular area of the scanned field.
- The value MN (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if MN is entered in starting character on the PR statement, the current value of MNUMBER is used at the time the PR statement is executed.

Because the value of MNUMBER may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation where the result field is invalid for the current operation only).

MODE Flag (GDBI Only)

The MODE flag  contains information about the application mode of operation. The following table shows the X and Y values. It is a character field with a length of 2 bytes.

In Operation Mode X =		In Update Mode Y =	
S	Standard Processing	R	Retrieval Update
M	MOSAIC Processing	U	Update
		L	Load Module

It is initialized prior to entering a mapping request. The information in the flag is generally required by database managers.

MSTART Flag

The MSTART flag is a text processing partial field flag that specifies the starting position of the middle (matching) part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value MS can be used in place of the standard partial field specification on the processing and record selection statement in order to access a particular area of the scanned field.
- The value MS (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if MS is entered in starting character on the PR statement, the current value of MSTART is used at the time the PR statement is executed.

Because the value of MSTART may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation where the result field is invalid for the current operation only).

MSTATUS Flag (GDBI Only)

The MSTATUS flag G is used by the mapping request to instruct VISION:Builder on a specific action to take subsequent to completion of the mapping request. It is a character field with a length of 6 bytes.

MSTATUS is initialized to blanks prior to each call to a mapping request. Permissible return values are listed in the following table.

Value	Explanation
blank	Mapping successfully completed.
NFOUND	Segment not found. Useful for input mapping only. Signifies to VISION:Builder that there was no data to fill the skeleton segment, either because of having run out of repeated segments, or because of failure to locate a keyed segment. In the process of building a hierarchical record (standard or MOSAIC processing, non-keyed operations), VISION:Builder returns to the mapping requests for each segment type to request repeated segments. Thus, the mapping requests must be programmed to issue NFOUND for each segment type to terminate a series of repeated segments.
STOPnn	Stop the run and set the condition code to nn (condition code setting support in OS/390 only). Used by mapping requests to cease processing.

The information is primarily useful for directing the operation of the mapping request, but can also be useful to a database manager.

OWN Flag

The OWN flag is used by own-code routines to communicate between own-code requests. It is a character field with a length of 16 bytes.

It is not examined or altered by VISION:Builder in any way. The flag name OWN can be entered in field name A, B, or C depending on the operation being performed.

PAGE Flag

The PAGE flag is used only in formatted reporting and specifies the placement of the page numbers, which are printed left-aligned in the title and summary lines of formatted reports. It is a character field with a length of 6 bytes. Enter the PAGE flag as F.PAGE.

PASSWORD Flag (GDBI Only)

The PASSWORD flag G contains the password as provided on the RF statement for the appropriate VISION:Builder file. It is a character field with a length of 8 bytes. It is available for all mapping requests, but is expected to be most valuable to the initialization mapping request to control access to the database. This field is set prior to a mapping request receiving control. Valid only in mapping requests.

RESTART Flag (IMS Only)

The RESTART flag I is used by requests to determine the restart status of a VISION:Builder run. It is a character field with a length of 8 bytes. If a run has been restarted, the RESTART flag contains the checkpoint ID at which the restart occurred. Otherwise, the value of the RESTART flag is all blanks.

RETURNCD Flag

The return code (RETURNCD flag) reflects the status of a CALL. It is a fixed field with a length of 4 bytes. The values are set by the called routine and the contents can be examined after a CALL. If the flag is included in a report, VISION:Builder treats the flag as a 2-byte fixed point field (output width of 7, maximum printable value 99,999). The value in this field is the value in general register 15 upon return to VISION:Builder from a called routine.

RNUMBER Flag

The RNUMBER flag is a text processing partial field flag that specifies the number of characters in the right part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value RN can be used in place of the standard partial field specification in order to access a particular area of the scanned field.
- The value RN (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if RN is entered in starting character on the PR statement, the current value of RNUMBER is used at the time the PR statement is executed.

Because the value of RNUMBER may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation where the result field is invalid for the current operation only).

ROW Flag

The ROW flag is set after successfully completing an array operation. It is a fixed field with a length of 4 bytes. It contains a numeric value set to the ROW number being processed. If the operation fails, the flag is set to zero. If an array operation is suppressed, the ROW flag is not suppressed.

RSTART Flag

The RSTART flag is a text processing partial field flag that specifies the starting position of the right part of the field. It is a fixed field with a length of 4 bytes. It is used in a scan left or right operation and can be tested, modified, or used in computation during request processing.

- The value RS can be used in place of the standard partial field specification on the processing and record selection statement in order to access a particular area of the scanned field.
- The value RS (dynamic partial field specification) cannot be used on the output specification (Rn) statement, nor in place of the flag names in operand A, operand B, or result specifications. For example, if RS is entered in starting character on the PR statement, the current value of RSTART is used at the time the PR statement is executed.

Because the value of RSTART may change during processing, you can use this flag to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag is invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid (except in a scan and substitute (SS) operation where the result field is invalid for the current operation only).

RSTATUS Flag

The RSTATUS flag indicates the results of the most recent request-read (RD, RE, or RG) operation performed in a PR statement. It is a character field with a length of 4 bytes.

The RSTATUS flag is set whenever a read (RD), read equal key (RE), or read greater than or equal key (RG) operator is used with coordinated files. The settings of the RSTATUS flag are shown in the following table.

Value	Explanation
AINV	Operand A contains an invalid field. The RE or RG operation was not performed.
AMIS	Operand A was missing. The RE or RG operation was not performed.
KEQL	The RE or RG operation was performed and a root segment with a key equal to the key in operand A was successfully returned, or an RD was successful.
KGTR	The RG operation was performed and a root segment with a key greater than the key in operand A was successfully returned.
NREC	The operation was performed but no record was found.
RIGN	The read operation was ignored. A segment in the specified coordinated file is in a loop or under control of a find segment (FS) operation.
EGRP	Specifies the end of a group. A sequential RD operator following a direct-read operator compares the key of each root segment retrieved to the search argument for the direct-read. When the key no longer matches, the RSTATUS flag indicates the end of group. EGRP is also set if end of file is reached on a sequential RD operation. In this situation, the EOF flag for the appropriate coordinated file is set to E and reset to Y on the next RE or RG operation for that file. This is the only situation in which VISION:Builder resets an EOF flag value. Successive RD operations without any intervening RE or RG operations continue to return the EGRP and EOF flag values.

An NS or GS statement immediately following an RE or RG operator causes an implied compare of the RSTATUS flag.

- If the RE or RG operation fails (that is, the RSTATUS flag is equal to AINV, AMIS, or NREC), the NS branch is taken.
- If the RE or RG operation is successful (that is, the RSTATUS flag is equal to KEQL, KGTR, or RIGN), processing continues with the next instruction.

You can use the GS statement in the same way as the NS statement to determine success or failure.

SEGNAME Flag (GDBI Only)

The SEGNAME field[G] contains the segment name as defined in columns [11-18] of the LS statement. It is a character field with a length of 8 bytes. If an alias segment name is provided on an LB statement in a run data group, the SEGNAME flag still refers to the segment name as defined on the LS statement.

The information is primarily useful for directing the operation of the mapping request, but can also be useful to a database manager. It is initialized prior to entering a mapping request.

SQL Flag (DB2 Only)

The SQL flag[D] contains status information after an attempt is made to insert a row into a DB2 table with a UNIQUE index. It is a character field with a length of 4 bytes. It contains the character string DUPL if a matching row already exists in the table. It contains blanks if a matching row does not exist.

- If F.SQL equals blanks, the record is successfully inserted.
- If F.SQL equals DUPL and Scan/Terminate does not equal 4, the run terminates; however, if Scan/Terminate equals 4, processing continues (the rejected row can be output to a report or another subfile). Note that this is the only instance in which the Scan/Terminate value determines processing based on message types other than type 4.
- If type 0 and type 1 messages are suppressed, there is no indication that the attempt to insert the row failed.

SSCOUNT Flag

The SSCOUNT flag counts the number of matches found during the REPLACE operation. It is a fixed field with a length of 2 bytes. The flag is set to zero at the start of each run. The settings are shown in the following table.

Setting	Result
0	There are no successful substitutes or the search-string is longer than the field to be modified.
-1	Used with variable length fields. If a REPLACE operation causes the maximum length of a type V field to be exceeded, the operation does not take place, and the field to be modified is made invalid.
-2	The field to be modified or search-string is null and/or invalid, and/or the replace-string is invalid prior to a REPLACE operation.

STRAN Flag (VISION:Builder 4000 Model Series Only)

The STRAN flag ❹ interrogates the status of the current transaction at each segment level and indicates the actions applied to the master file record by the current transaction. It is a character field with a length of 9 bytes. The STRAN flag is used in type M requests and their subroutines, but is available to any request type.

Each of the 9 bytes of the STRAN flag corresponds to a segment level in a structured record. The settings are shown in the following table.

Setting	Explanation
Blank	No action at this level.
M	Match actions only occurred at this level. All match fields for the segment at this level were matched (transaction action code M), but no fields in the segment were updated (action codes B, R, A, S, P, or X).
I	A segment was inserted at this level. This setting applies to level 1 (root), as well as lower levels 2 through 9. Thus, action code C or I was used or M with default create or insert specified. In addition, update actions (codes B, R, A, S, P, or X) may have been applied after the create or insert.
U	Update actions (codes B, R, A, S, P, or X) were applied to the segment at this level, but C or I was not specified.
D	The segment at this level was deleted by a transaction (action code E). This setting applies only to the level that was explicitly deleted by the transaction, not to lower level dependent segments. This setting is not apparent at level 1, because deleted master records are not presented to any type of request.

The STRAN flag is reset to blanks before obtaining the next logical transaction record (that is, either reading a new transaction record or retrying the last one). However, the STRAN flag is not revalidated if you set it to an invalid value. This is consistent with the treatment of other user flags.

The STRAN flag is set to non-blank values as the transaction actions are successfully applied.

If a transaction is referenced in the type 4 request and rejected because of update errors, the STRAN flag reflects the match actions that occurred but does not reflect the insert, update, or delete actions. These actions were either not yet applied or were backed out before the type 4 request received control.

TIME Flag

The TIME flag records the time of day at which the run was started. It is a character field with a length of 8 bytes. The TIME flag prints as HH.MM.SS, where HH equals hours, MM equals minutes, and SS equals seconds. Leading zeroes are added where necessary.

TODAY Flag

The TODAY flag records the system date in two formats.

Note: The order of month, day, and year and the date delimiter (/) are installation options and can be changed using M4PARAMS. See the *Installation Guide*.

Normal processing/standard reporting (6 characters)

The TODAY flag records the date in the format MMDDYY, where MM equals 2 numerals for month, DD equals 2 numerals for day, and YY equals 2 numerals for year (that is, January 15, 2001 equals 011501).

Formatted reporting (8 characters)

The TODAY flag records the date in the format MM/DD/YY (01/15/01).

TODAYX Flag

The TODAYX flag records the system date in two formats.

Note: The order of month, day, and year and the date delimiter (/) are installation options and can be changed using M4PARAMS. See the *Installation Guide*.

Normal processing/standard reporting (8 characters)

The TODAYX flag records the date in the format MMDDYYYY, where MM equals 2 numerals for month, DD equals 2 numerals for day, and YYYY equals 4 numerals for year (that is, January 15, 2001 equals 01152001).

Formatted reporting (10 characters)

The TODAYX flag records the date in the format MM/DD/YYYY (01/15/2001).

TRAN Flag (VISION:Builder 4000 Model Series Only)

The TRAN flag ❹ indicates the maintenance status of a master file record and/or the rejection of a transaction record. It is a fixed field with a length of 1 byte. The settings are shown in the following table.

Value	Conditions
0	No transaction records exist for the master file record.
1	The master file record was updated or matched by a transaction record.
2	The master file record was created by a transaction record.
3	Conditions for values 1 and 2 both apply.
4	One or more transaction records against this master file record were rejected.
5	Conditions for values 1 and 4 both apply.
6	Conditions for values 2 and 4 both apply.
7	Conditions for values 1, 2, and 4 all apply.
8 or higher	Invalid.

XTRAN Flag (VISION:Builder 4000 Model Series Only)

The XTRAN flag ❹ identifies the reasons for rejected transactions. It is a fixed field with a length of 1 byte. When more than one error is detected in a transaction, only the last one is described.

The XTRAN settings are shown in the following table.

Value	Explanation
0	Inactive setting.
1	Request rejection.
2	The transaction record does not match any of the defined identifiers.
3	The field in the transaction is outside the maximum or minimum values specified in the transaction definition.
4	A transaction record key is less than the preceding transaction record key; the transaction file is out of sequence.
5	Transaction record key field with M, D, or C action cannot be converted for matching purposes.

Value	Explanation <$paranum
6	The contents of the date field in the transaction do not meet the date validation criteria.
7	The field in the transaction record does not conform to the input edit pattern specified in the transaction definition.
8	The transaction record key does not match any master file key and a create is not specified.
9	A create action has been specified for a record that already exists on the master file.
10	Field associated with an M, E, C, D, or I action cannot be converted for matching purposes or a field associated with B, R, A, S, P, or X action causes an arithmetic or conversion error.
11	A non-record key field (that is, any field other than a record key field) associated with an M action cannot be located in the master file record.
12	The segment for which a delete segment action (E) is specified does not exist in the master file record.
13	The insert action (I) has been specified for a segment that is repeated a fixed number of times. There are no “empty” segments into which the insert can be made.
14	A segment to be inserted already exists in the master file record.
15	This value occurs with variable length records only. The requested insertion would cause the maximum record length to be exceeded.
17	The count field controlling the segment to be inserted is too small to contain the number of subordinate segments being inserted.

Appendix

D ASL Reference Summary

For information on IBM® Language Environment® Callable Services (CEExxx), see the *IBM Language Environment for MVS® and VM Programming Reference*.

Terminology

Term	Description
Procedure	A procedure defines an algorithm or sequence of calculations. It is composed of a series of procedure statements.
Procedure statement	A procedure statement begins on a new line and consists of an optional label followed by a command.
Label	A label identifies a specific statement. The label is optional. If used, place the label before a procedure statement and follow the label immediatedly (without intervening spaces) by a colon (:).
Command	A command identifies the kind of procedure statement. You can follow a command by keywords, functions, constants, names, qualifiers, expressions, and comments.
Function	A function is a subrountine that derives a value from other data values.
Keyword	A keyword identifies how certain parameters are used. Most keywords are optional.
Keyword operand	A keyword operand identifies the data values associated (or just operand) with a keyword. In the procedure statement, follow a keyword with one or more spaces and a keyword operand.
Keyword phrase	A keyword phrase is a keyword plus its operand.

Constants

There are six types of constants: character, integer, decimal, floating point, time, and pattern. A description of each follows:

- Delimit **character constants** by single quotation marks. Specify a single quotation mark within a constant by two consecutive single quotation marks. For example: 'A' or 'CAN''T'
- Specify **integer constants** with numeric digits only. If the integer constant is negative, place a minus sign (-) before the constant. If the integer constant is positive, you can place an optional plus sign (+) before the constant.

  For example: -2, 0, or +7
- Specify **decimal constants** with numeric digits, an optional sign, and a decimal point. If you use a plus sign (+) or a minus sign (-), make it the first character of the constant. For example: -2000.00, 3.99, or +7.25
- Specify **floating point constants** with an optional sign preceding an integer or decimal constant followed by a power of ten expressed in exponential notation. For example: 1.50E10 or 13.75E-5
- Specify a **time constant**, HH:MM:SS.nn...n (hours, minutes, seconds, decimal seconds) by the letter T, followed by a single quotation mark, the time constant, and a closing single quotation mark.

  For example: T'12:01:00.125'
- Specify a **pattern starting** with the letter P, followed by a single quotation mark, a string of special pattern symbols, and a closing single quotation mark. For example: P'#(#999#)#999#-#9999' or P'ZZZ999'

Names

- Begin field names and statement labels with an alphabetic character (A-Z).
- Make the remaining seven characters either alphabetic characters (A-Z), numeric digits (0-9), or underscores (_).
- Enclose field names that do not conform to this syntax in double quotation marks.

Qualifiers

You can prefix a field name with a standard 1-character qualifier and a period. A qualifier identifies the type of field or file where the field exists. The following are valid qualifiers and their meanings:

Qualifier	Type of Location	
Blank or N	New master file.	
1-9	Coordinated files 1-9.	
T	Temporary file.	
F	Flag field.	
X	Transaction file	For VISION:Builder and VISION:Two, but not VISION:Inform.
O or 0	Old master file.	
W	Working storage	
V	Linkage section.	
A, B, E, H, J K, M, Q, 1-9	Array (must match the qualifier that identifies the array.	

Comments

Place comments anywhere following a semicolon (;), except on a continued line.

Arithmetic Expressions

Code arithmetic expressions with the operators +, -, *, and / for addition, subtraction, multiplication, and division, respectively.

For example: A + B A - B B * C D / C

As in conventional algebraic notation, operations within an arithmetic expression are processeded according to a specific hierarchy, from left to right. However, multiplication and division are performed prior to addition and subtraction unless this order is overridden by parentheses.

Logical Expressions

Use logical expressions in IF, CASE, and DO statements. Make logical expressions from conditional functions, relational expressions, or list expressions connected by logical operators.

- A conditional function is a function that returns a true or false condition.
- A relational expression is composed of two values connected by a logical operator. In relational expressions, you can represent the logical operator as characters or as symbols:

  EQ or =
  NE or <>
  GT or >
  LT or <
  GE or >=
  LE or <=

- A list expression lists the specific values to test.

  For example: NUMBER = 0001 0002 0003

Statement Syntax

The following describes the conventions used to provide a precise description of the syntax of a function or command. Enter commands and functions in the exact order given.

- Brackets [ ] indicate an optional parameter.
- Braces { } indicate a choice of entry. Unless a default is indicated, you must choose one of the entries.
- Required parameters do not have brackets or braces surrounding them.
- Items separated by a vertical bar (|) represent alternative items. Select only one of the items.
- An ellipsis (...) indicates that you can use a progressive sequence of the type immediately preceding the ellipsis. For example: name1, name2, ...
- Uppercase type indicates the characters to be entered. Enter such items exactly as illustrated. Sometimes you can use an abbreviation.
- Lowercase, italic type specifies fields to be supplied by the user.
- Separate commands, keywords, and keyword phrases by blanks.
- Enter punctuation exactly as shown (parentheses, colons, and so on).

Continuation

You can write procedure statements on multiple lines. Terminate each line by a blank space followed by a comma. Continue the remainder of the statement on the following lines. For example:

```
IF       NAME    = 'THE ABC COMPANY ' ,
 AND NUMBER = '00001' ,
 OR   NUMBER = '0002' ,
 OR   NUMBER = '0003' ,
THEN
```

Built-in Functions

Specify functions by entering a function name followed immediately (with no intervening spaces) by a left parenthesis, one or more keyword phrases, and terminating with a right parenthesis.

Conditional Functions

Conditional functions return a true or false condition.

FIND([SEGMENT] *segment-name* [FIRST | LAST | NEXT |
WHERE *selection-expression*])

LOCATE([ARRAY] *array-identifier* { [ROW *row-number*]
[COLUMN *column-number*] })

SCAN([FIELD] *field-name* [FOR] *search-value/pattern*
{ FROM LEFT | FROM RIGHT } [NOTEQUAL])

VALIDATE([FIELD] *field-name* { PATTERN P'*pattern*' | DATE })

Value Functions

Value functions return an actual value, either a result from a table or part of an existing field.

LOOKUP([TABLE] *table-name* [ARGUMENT] *lookup-argument*
[NEAREST | SMALLER | LARGER | INTERPOLATE])

PF([FIELD] *field-name* [START] *start-position*
[LENGTH *partial-length*])

Commands

Long commands that have abbreviations are shown in braces { } to indicate choice. For example: { COMBINE | COM }, { CONTINUE | CONT }, { FIELD | FLD }, and so on.

```
CALL{[PROCEDURE] procedure-name |
        REPORT report-name |
        SUBFILE subfile-name |
        MODULE 'module-name' |
        CEEDATE | CEEDATM | CEEDAYS |
        CEEDYWK | CEEGMT |CEEGMTO |
        CEEISEC | CEELOCT | CEEQCEN |
        CEESCEN |CEESECI | CEESECS |
        CEEUTC
        [USING parm ...] }
```

The IBM Language Environment (LE) Calllable Services (CEExxx routines) are Year 2000 compliant.

```
CASE   [ WHERE ] logical-expression
```

```
COLLATE  {REPORTS report-name ...
            [ KEYLENGTH length] |
            ALL KEYLENGTH length }
```

For VISION:Builder and VISION:Two.

```
{ COMBINE | COM } [ FIELDS ] field1 ... STORE result-field
        [ [ BLANKS ] number ]
```

```
{ CONTINUE | CONT }
```

```
DO   {[WHILE logical-expression]
      [UNTIL logical-expression]
      [FORALL segment-name]
      {[FORALL CELLS IN ARRAY array-identifier]
      [FORALL COLUMNS IN ARRAY array-identifier
            [WITHIN ROW row-number]]
      [FORALL ROWS IN ARRAY array-identifier
            [WITHIN COLUMN column-number]]
      [FOR integer]  }  |
      [CASE]
```

Array keywords are available in VISION:Builder and VISION:Two, but not VISION:Inform.

```
ELSE

END
```

field name: { FIELD | FLD } [TYPE] *field-type* [[LENGTH] *field-length*]
[DECIMALS *decimal-places*]
[FLOAT *floating-edit-char*]
[FILL *fill-edit-char*]
[TRAIL *trailing-edit-char*]
[EDLEN *edit-length*]
[INIT *initial-value*]
[HEADING *'line1'* [*'line2'*]]

GO [TO] *jump-to-label*

IF [CONDITION] *logical-expression* [THEN]

LEAVE

LET [FIELD] *result-field = source-expression*
[WITH] [EDIT P'*pattern*'] [ROUNDING]
[JUSTIFY { LEFT | RIGHT }]

{LOCATE | LOC}
[ARRAY] *array-identifier*
{[ROW *row-number*]
[COLUMN *column-number*]}

} For VISION:Builder and VISION:Two.

{ RELEASE | REL } { [SEGMENT] segment-name | ARRAY array-identifier }

{ REPLACE | REP } [STRING] *search-string* [IN] *modify-field*
[WITH] *substitute-value*

{ RETURN | RET }

ROUTE { REPORTS *report-name* ... | ALL }
[KEYVALUE *'data-value'*]
TO *destination* ... [DEFER]

} For VISION:Builder and VISION:Two.

ROUTE { REPORTS *report-name* ... | ALL }
[KEYVALUE *'data-value'*]
TO *destination* ... [DEFER]

TRANSFER [TO] { NEXT_MASTER | TYPE_1 | TYPE_2 }

Contacting Technical Support

For technical assistance with this product, contact Computer Associates Technical Support on the Internet at esupport.ca.com. Technical support is available 24 hours a day, 7 days a week.

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X