

Advantage™ VISION:Transact® for CICS®

ASL Reference Guide

Release 7.8



Computer Associates®

B02568-1E

This documentation and related computer software program (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by Computer Associates International, Inc. ("CA") at any time.

This documentation may not be copied, transferred, reproduced, disclosed or duplicated, in whole or in part, without the prior written consent of CA. This documentation is proprietary information of CA and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of this documentation for their own internal use, provided that all CA copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the confidentiality provisions of the license for the software are permitted to have access to such copies.

This right to print copies is limited to the period during which the license for the product remains in full force and effect. Should the license terminate for any reason, it shall be the user's responsibility to return to CA the reproduced copies or to certify to CA that same have been destroyed.

To the extent permitted by applicable law, CA provides this documentation "as is" without warranty of any kind, including without limitation, any implied warranties of merchantability, fitness for a particular purpose or noninfringement. In no event will CA be liable to the end user or any third party for any loss or damage, direct or indirect, from the use of this documentation, including without limitation, lost profits, business interruption, goodwill, or lost data, even if CA is expressly advised of such loss or damage.

The use of any product referenced in this documentation and this documentation is governed by the end user's applicable license agreement.

The manufacturer of this documentation is Computer Associates International, Inc.

Provided with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227- 19(c)(1) and (2) or DFARS Section 252.227-7013(c)(1)(ii) or applicable successor provisions.

© 2004 Computer Associates International, Inc.

All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contents

Chapter 1: Terminology, Syntax, and Processing

Introduction	1-1
ASL Terminology	1-2
Syntax Terminology	1-2
Constants	1-3
Character Constants	1-3
Integer Constants	1-4
Decimal Constants	1-4
Floating Point Constants	1-4
Patterns	1-4
Names	1-5
Comments	1-5
Arithmetic Expressions	1-6
Logical Expressions	1-7
Statement Syntax	1-9
Statement Layout	1-9
CALL Command	1-10
The Nature of ASL	1-11
Implicit Loops and Set Operation	1-11
Record and Segment Processing	1-16
Conclusion	1-16
Defining ASL Procedures	1-17

Chapter 2: Built-In Function

What Is a Built-In Function?	2-1
PF Function	2-1

Chapter 3: Procedure Statements

Overview of Statements	3-1
Syntax and Examples	3-2
CALL Command	3-2
CASE Command	3-4
CONTINUE Command	3-5
DO Command	3-5
ELSE Command	3-9
END Command	3-10
FIELD Command	3-11
GO Command	3-12
IF Command	3-13
LEAVE Command	3-14
LET Command	3-15
MSG Command	3-16
RETURN Command	3-17
SET Command	3-18
TRACE Command	3-19
TRANSFER Command	3-20

Chapter 4: ASL Examples

Appendix A: Relationship of ASL Statements to VISION:Transact

Appendix B: Technical Notes

Reserved Words	B-2
Qualifiers	B-4
PATTERNS	B-4
Valid Entries for Field Type and Length	B-5

Appendix C: Flags

TODAY, DATE, ISDATE, and JULIAN Flags	C-3
TIME Flag	C-3
CSTATUS and Returned Flags	C-4
ATTNID Flag	C-5

ERROR Flag	C-5
TRANCODE Flag	C-6
GOSC Flag	C-6
TERMID, USERID, OPERID, and OPERCL Flags	C-7
MSGLINE Flag	C-7
Dynamic Partial Field Flags	C-8
HIGHVALU and LOWVALU Flags	C-8
PAGENO Flag	C-9
SCRNAMEI and SCRNAMEO Flags	C-9
Execution Tracing Flags	C-10

Index

Terminology, Syntax, and Processing

Introduction

ASL (Advanced Syntax Language) 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.

This guide contains information specific to the use and operation of ASL and assumes that you are familiar with VISION:Transact.

- [Chapter 1, “Terminology, Syntax, and Processing”](#) has information about how to read the syntax of the statements and defines some common computer terminology with respect to ASL.
- [Chapter 2, “Built-In Function”](#) describes the built-in functions of ASL. Built-in functions are used as a part of other procedure statements. The syntax of the function is given first, then the description of the function, followed by descriptions of each keyword phrase.
- [Chapter 3, “Procedure Statements”](#) describes the procedure statements. Again, the syntax is displayed and defined, and then examples are given to illustrate the procedure statement.
- [Chapter 4, “ASL Examples”](#) contains examples of ASL usage showing the use of the language in the context of an application or procedure.
- [Appendix A, “Relationship of ASL Statements to VISION:Transact”](#) relates the functions and procedure statements to their VISION:Transact 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”](#) describes the VISION:Transact flag fields.

ASL Terminology

The purpose of this chapter is to define the terms that are used to describe ASL statements.

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 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 FIELD A = FIELD B
```

is an actual procedure statement where:

LABELX	The statement label.
LET	The command
FIELD A=FIELD B	Places the contents of field B into field A.

Syntax Terminology

We use the following terminology to describe ASL.:

Field	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 and consists of an optional label followed by a command.
Label	A label identifies a specific statement. It is optional at the beginning of a procedure statement and must be followed immediately by a colon (:).
Command	A command identifies the kind of procedure statement. A command can include keywords, functions, constants, names, expressions, and comments.
Function	A function is a value that is derived from other data values.
Keyword	A keyword identifies how certain parameters are used. Most keywords are optional

Field	Description
Keyword operand	A keyword operand (or operand) identifies the data values associated with a keyword. Keyword operands immediately follow a keyword (if used) in the procedure statement
Keyword phrase	A keyword phrase is the name given to the combination of a keyword plus its operand

In general, one or more spaces must separate the command, keywords, and operands.

The following is an example of an ASL command using blanks as separators between the keywords and operands on the FIELD command.

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

Procedure statements can be written using multiple lines. Each line (ignoring comments) must be terminated by a comma preceded by a blank and continued on the next line. 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 five types of constants: character, integer, decimal, floating point, and pattern. Each type of constant is described in the following pages.

Character Constants

Character constants are delimited by single quotation marks. For example:

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

Two consecutive single quotation marks within the constant are used to represent a single quotation mark.

Integer Constants

Integer constants contain only numeric digit(s) and a minus sign if the integer is negative. A plus sign for positive numbers is optional. The sign, if used, must be the first character after the constant. For example:

```
+100  
-2  
0
```

Decimal Constants

Decimal constants can contain numeric digit(s), an optional sign, and a decimal point. When used, the sign + or – must be the first character of the constant. For example:

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

Floating Point Constants

Floating point constants consist of 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
```

Patterns

A pattern is a special name representing a numeric format that must be preceded by the letter P and a period. For example:

```
P.MMDDYY  
P.NUMERIC
```

For more information on patterns, see [Appendix B, “Technical Notes”](#).

Names

Field names and statement labels can consist of up to eight characters, the first of which must be an alphabetic character (A-Z). The remaining characters can either be alphabetic characters (A-Z), numeric digits (0-9), or underscores (_). The following are valid field names or procedure statement labels:

```
CUST_NUM  
BAL30  
CUSTNO
```

A standard one-character qualifier and a period can prefix a name. A qualifier identifies a specific file or usage of a field. The following are valid *qualified* field names:

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

Qualifiers relate fields to their origins. For example:

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

[Appendix B, “Technical Notes”](#) contains a list of all valid qualifiers and their origins.

Names that include special characters must be enclosed 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 would make it appear to be two separate fields instead of one.

Comments

Comments can be placed anywhere following a semicolon (;), except on a continued line.

```
;      RESETING 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. The comment is separated from the command and its keyword phrase by the semicolon.

Arithmetic Expressions

Arithmetic is performed by coding 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 according to a specific hierarchy, basically from left to right. However, multiplication and division are performed prior to addition and subtraction unless this order is overridden by parentheses.

A-B*C+D

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

When the expression is written as:

(A-B) * (C+D)

the addition and subtraction are done 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. The following steps are taken to come up with the result of the arithmetic expressions.

Step	Without Parenthesis	With Parenthesis
1.	A-B*C+D	(A-B)*(C+D)
2.	5-3*2+1	(5-3)*(2+1)
3.	5-6+1	2 * 3
4.	0	6

Logical Expressions

Logical expressions are used in **IF**, **CASE**, and **DO** commands.

A logical expression consists of relational expressions or list expressions connected by logical operators.

In the following examples, A can be a field name, and B and C can be any constant, field name, or arithmetic expression. B can also be a pattern if the operator is EQ or NE.

A relational expression takes one of the following forms:

- 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)

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 relational expressions the logical operator can be represented as one of the following:

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

There are three Boolean logical operators: AND, OR, and NOT. Parentheses may be used 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 logical expression.

The NOT operator also can be used to *exclude* a few values that are not wanted for processing instead of *including* all of the values that are wanted for processing. For example:

NOT (NUMBER = 1 OR NUMBER = 5)

is true for all values of NUMBER except 1 or 5.

[Figure 1-1](#) illustrates numeric examples and the subsequent true or false evaluation of the 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

Figure 1-1 Logical Expressions

Statement Syntax

The following describes the conventions used to provide a precise description of the syntax of a function or command. The syntax is displayed at the top of each page beginning a new function or command.

You must enter commands and functions in the *exact order* given on the syntax line.

The following syntax conventions are for the purpose of defining the syntax of the ASL statements. As the examples given with each command show, you normally enter your procedures using upper-case 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 will not have brackets or braces surrounding them.
- Items separated by a vertical bar (|) represent alternative items. No more than one of the items may be selected.
- An ellipsis (...) indicates that multiple entries of the type immediately preceding the ellipsis are allowed.
- **Boldface** type indicates the characters to be entered. Such items must be entered exactly as illustrated. Sometimes an abbreviation can be used.
- *Italic* type specifies fields to be supplied by the user.
- Underscored type indicates a default option. If the parameter is omitted, the underscored value is assumed.
- Commands, keywords, and keyword phrases must be separated by blanks.
- Punctuation must be entered exactly as shown (parentheses, colons, etc.).

Statement Layout

The following describes the format used to present each function or procedure statement. Use the sample page layout on the following page for reference to the six rules.

1. The name of the command appears in bold at the left of the page.
2. The format of the procedure statement is presented immediately below the command, indented.
3. The purpose for each procedure statement is described below its format.
4. The keywords in the procedure statement are explained following the purpose of the procedure statement.

5. Abbreviations for procedure statements or keywords appear directly beneath the procedure statement or keyword.
6. Examples using the procedure statement follow the descriptions of its parts.

CALL Command 1

2 `CALL {[PROCEDURE] procedure-name |
MODULE module-name [USING parm1 ...] [PLI] |
DLI USING parm1 ...}`

- 3 The **CALL** command branches to an internal procedure or an external user-written module.

PROCEDURE *procedure-name*

- 4 **PROC** Specifies a subroutine procedure to be executed. The operand must be the name of a subroutine procedure within the current application

- 5 **MODULE** *module-name*

MOD Specifies an external user-written load module to call. The operand (*module-name*) must be an external load module name

USING *parm 1...* Is an optional entry that specifies any parameters to be passed to the **MODULE**. The operands can be field names, constants, or arithmetic expressions. For CICS applications, refer to the Environment Manual to determine whether or not you should specify parameters on the CALL command

PLI Indicates that the **MODULE** named in the keyword phrase is written in PL/I. This keyword has no operands and is used only with IMS TM applications. Do not enter this keyword when the external user-written program is not coded in PL/I

DLI USING *parm1...*

Indicates a standard DL/I call. This keyword is used with IMS or CICS applications. The USING parameter must follow the IBM conventions for a DL/I call

Example 1:

```
CALL DLI USING 'GN ' V.PCB01 W.SEGIO
```

- 6 This example calls the DLI interface program ASMTDLI with parameters for the function code, PCB, and I/O area.

The Nature of ASL

ASL has been designed to enable the application developer to work with a logically related set of data. The objective behind ASL is to enable the developer to focus upon the algorithms and business consideration 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 which enable the developer to process his data correctly and consistently.

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

There are two important and powerful capabilities built into the ASL processor and the ASL language which the application designer should be aware of. These are:

1. Implicit loops or set operation.
2. 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 (i.e., 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 wishes 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. Thus, the application designer 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 1-2](#).

Field Name	DEPT	MANAGER	MGRSAL	Level 1
	SALES	GREEN	60,000	

Field Name	NAME	SALARY	SALGRADE	Level 2
	SMITH	40,000	5	
	JONES	50,000	6	
	WHITE	40,000	5	
	BROWN	65,000	7	

Figure 1-2 Department Record With Subordinate Segments

This structure is typical of a department record with subordinate segments, one for each employee. Let us suppose that we wish to increase the salary of everybody in the department by 5%. This is achieved by 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 1-3](#).

Field Name	DEPT	MANAGER	MGRSAL	Level 1
	SALES	GREEN	60,000	

Field Name	NAME	SALARY	SALGRADE	Level 2
	SMITH	42,000	5	
	JONES	52,500	6	
	WHITE	42,000	5	
	BROWN	68,250	7	

Figure 1-3 Department With Updated Subordinate Segments

The single statement acted on all occurrences of the data.

Let us suppose we have our original data of [Figure 1-2](#), but we only want to increase the salaries of those people whose salary grade is less than 7. The ASL statements shown below achieve this.

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

The result of this statement is shown in [Figure 1-4](#).

Field Name	DEPT	MANAGER	MGRSAL	Level 1
	SALES	GREEN	60,000	

Field Name	NAME	SALARY	SALGRADE	Level 2
	SMITH	42,000	5	
	JONES	52,500	6	
	WHITE	42,000	5	
	BROWN	65,000	7	

Figure 1-4 Selection Logic

The ASL processor identifies all segments of data that are relevant and executes the statement. The application developer doesn't have to write any complex looping structures.

As another example, let us assume that the three-level record structure shown in [Figure 1-5](#) exists.

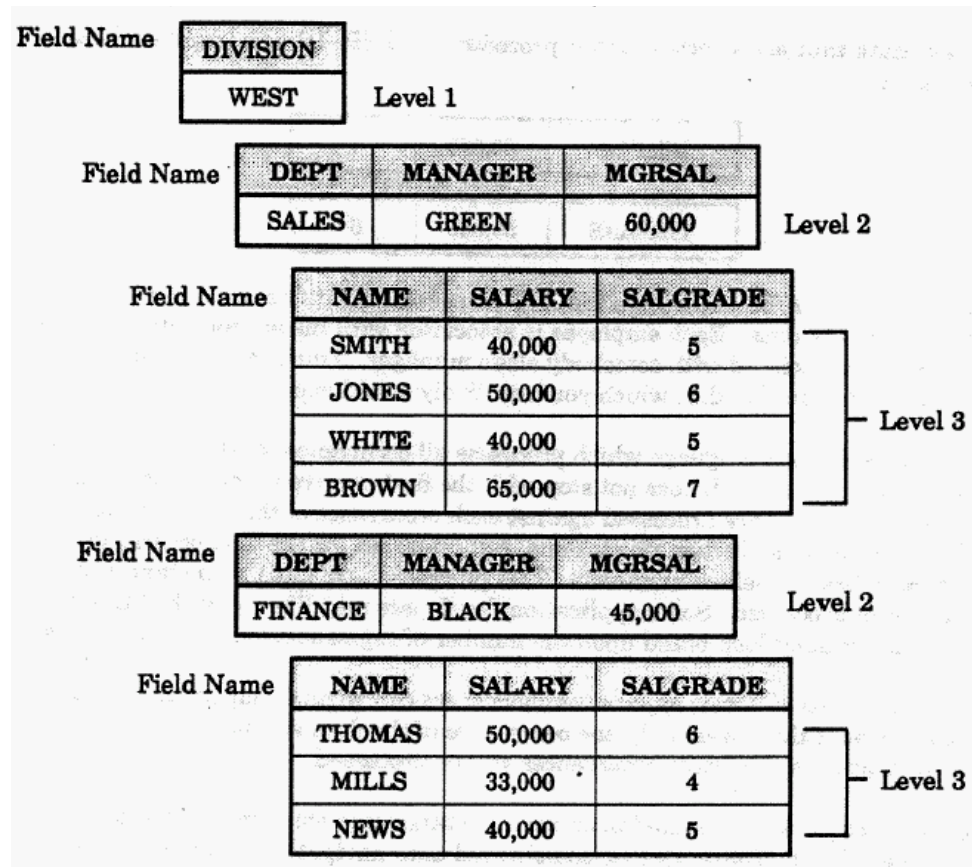


Figure 1-5 Three-level Data Structure

We want to select every employee who earns more than his manager. The ASL statements shown below achieve this.

```

IF SALARY GT MGRSAL
  CALL PROCEDURE HIGHPAID
END

```

The sets of data that are selected for the procedure HIGHPAID as a result of these ASL statements are:

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 his appropriate manager. No employee is associated with somebody else's manager. Thus, the data that is returned and processed is that which you intuitively would expect.

ASL, therefore, is a set language which processes all occurrences of the data that satisfy any criteria that exist. 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 and 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.

In our examples we simply stated in one or two statements what we wanted to do. The ASL processor did it. There were no complicated data navigation commands; it was all transparent. Focus on the problem and leave the data to ASL.

In some sophisticated instances, it is necessary for an application developer to know that he is in fact dealing with a set of data and not an individual occurrence. Under some circumstances, two specific situations can occur:

1. The developer wants to stop the procedural algorithm working on the particular occurrence of the data and move on to the next occurrence.
2. The user wants to 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 they are important performance considerations.

When processing large structured records, for example, there may be a considerable performance improvement by bypassing processing of segments where it is known that no action is required.

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 record for the current procedure.

Within ASL, any selection statement (**IF**, **DO**, etc.) limits the available segments of data for processing by subsequent procedural statements. This applies also to subroutine procedures and output screen processing. 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 as described above is a philosophy that says 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 handling of 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 is a rather 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, the application developer can write his application as if he were dealing with all the records as if they were in memory. 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 needs to be made in the case of browsing. All of this is transparent to the application designer.

Conclusion

ASL is a powerful language. It enables the application designer to focus on the business problem at hand without worrying about the problems of data access and data navigation.

ASL provides powerful commands for those sophisticated applications where the developer has a need to control the navigation through the data.

Defining ASL Procedures

You can define ASL procedures through the VISION:Workbench for ISPF. ASL can also be used in place of or in addition to traditional VISION:Transact requests.

The VISION:Workbench for ISPF has a Procedure Definition screen on which you can enter ASL procedures. A sample screen is shown in [Figure 1-6](#).

```

ASL  ---- TRANSACT.WBISPF.M5INPUT (APPL2)  ----- ROW 1 OF 31
COMMAND ==>                                SCROLL ==> CSR
                                           PROCEDURE: ASLPROC

      Procedure Type ==> _   Screen Name ==> _____
      Reinit Temps?  ==> _

Line  Free Form Processing Statements
Cmd  -----
'''  IF QANONHND < 5
'''    CALL PROCEDURE REORDER
'''  END
'''
'''  _____
'''  _____
'''  _____

```

Figure 1-6 VISION:Workbench for ISPF Screen

Although ASL is generally used through the VISION:Workbench for ISPF, it can also be entered instream in VISION:Transact requests. ASL statements are entered after an ER statement. The next line must contain two system delimiters followed by **PROC**, starting in column one. Then you enter the ASL procedure. The line after all of the ASL statements for the procedure must contain two system delimiters followed by **PEND**, starting in column one. The following 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 PROCEDURE STOCKS
END
##PEND

```

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

Built-In Function

What Is a Built-In Function?

A function is a value that does not actually exist anywhere but is created when reference is made to it. Although functions do not exist until they are needed, they can be thought of as if they are real fields. Functions are written differently than fields, but basically they can be used wherever a field name can be used.

A function represents a field value. A function, when needed, is used within procedure statements and can be used anywhere a field name can be used. There is one built-in function available in ASL: the partial field function.

The partial field function is specified by using the function name followed immediately by a left parenthesis, one or more keyword phrases, and terminated with a right parenthesis. The partial field function is described in the following pages.

PF Function

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. The **PF** function can be used on a character field and be used in any replacement or logical expression. There are three keywords: FIELD, START, and LENGTH.

FIELD *field-name* Specifies the name of a character field. The **PF** function returns a value which is only part of the field specified in this operand

START *start-position*

Is a required entry which 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 the flag field name **PS**, where **PS** is the value of the starting location of the field you are partial fielding

LENGTH *partial-length*

LEN Is a required entry. 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 the flag field name **PN**, where **PN** is the length of the field you are partial fielding.

Example 1:

Assume these values for the following fields:

```
CHARS      = 'ABCDEFGHIJKL'
T.NUMBER   = '3'
PS         = '2'
PN         = '6'
```

Function	Partial Field Value
PF(CHARS START 1 LENGTH 7)	'ABCDEFG'
PF(CHARS 3 PN)	'CDEFGH'
PF(CHARS PS 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:

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

Field	Field Contents
CUSTNO	'1000'
PF(CUSTNO START 1 LENGTH 1)	'1'

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.

Procedure Statements

Overview of Statements

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

CALL	Calls another procedure or an external user-written module.
CASE	Begins a group of procedure statements within a DO CASE block which are to be performed when the CASE condition is true.
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 which 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
MSG	Terminates the interaction and displays a message when screen field validation fails.
RETURN RET	Returns control to the calling procedure.
SET	Specifies the screen field to be processed by error handling.

TRACE	Displays literals or fields during application execution.
TRANSFER	Causes control to be transferred to a specific point within the processing cycle

Syntax and Examples

Each statement is described in detail on the following pages. 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.

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

Field	Contents
CUSTNO	'00001'

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

See [Chapter 1, “Terminology, Syntax, and Processing”](#) for the rules and explanations of notational conventions and page layout.

CALL Command

```
CALL {[PROCEDURE] procedure-name |  
      MODULE module-name [USING parm1 ...] [PLI] |}  
      DLI USING parm1 ...}
```

The **CALL** command branches to an internal procedure or an external user-written module.

PROCEDURE *procedure-name*

PROC Specifies a subroutine procedure to be executed. The operand must be the name of a subroutine procedure within the current application

MODULE *module-name*

MOD Specifies an external user-written load module to call. The operand (*module-name*) must be an external load module name

USING *parm 1...* An optional entry that specifies any parameters to be passed to the **MODULE**. The operands can be field names, constants, or arithmetic expressions. For CICS applications, refer to the Environment Manual to determine whether or not you should specify parameters on the CALL command.

PLI Indicates that the **MODULE** named in the keyword phrase is written in PL/I. This keyword has no operands and is used only with IMS TM applications. Do not enter this keyword when the external user-written program is not coded in PL/I

DLI USING *parm1...*

Indicates a standard DL/I call. This keyword is used with IMS or CICS applications. The USING parameter must follow the IBM conventions for a DL/I call

Example 1:

```
CALL DLI USING 'GN ' V.PCB01 W.SEGIO
```

This example calls the DLI interface program ASMTDLI with parameters for the function code, PCB, and I/O area.

Example 2:

```
CALL MODULE PLICODE USING T.PARM1 T.PARM2 PLI
```

This IMS TM example calls an external program PLICODE that is written in PL/I. Two parameters are being passed in the temporary fields T.PARM1 and T.PARM2.

Example 3:

```
CALL MODULE CONVERT USING W.COMMAREA T.LENGTH
```

This CICS example calls an external program CONVERT with two parameters specified. For CICS applications, two parameters on the CALL command indicate that the data is passed through the CICS COMMAREA. The first parameter (W.COMMAREA) names the field that defines the first byte of the COMMAREA. The second parameter (T.LENGTH) defines a two-byte fixed point binary field that contains the length of the COMMAREA to be passed to the called program.

Example 4:

```
CALL PROCEDURE SELECT
```

In this example, a branch to the internal subroutine procedure SELECT is being made. After SELECT is finished, control is passed to the statement after the **CALL**.

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.

Only the *first* group of procedure statements whose **CASE** command condition is true is performed within the **DO CASE** block. Once the group of procedure statements has been executed, 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 is passed to the statement immediately following the **CASE** command; otherwise, control is passed to the next **CASE**, **ELSE**, or **END** command within the current **DO CASE** block

Example 1:

```
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.TEMP1 = -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 (i.e., CUSTNO is less than 00001)

CONTINUE Command

CONTINUE The **CONTINUE** command causes the processing cycle to continue with the next occurrence of the segment controlling the record domain.

Note that this action is equivalent to a **RETURN** 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 PROCEDURE LISTX
ELSE
    CONTINUE
END
CALL PROCEDURE LAST
```

A CUSTNO value of 00001 causes the **LET** command to execute followed by the **CALL** to the procedure LISTX. Then the **CALL** to the procedure LAST is executed.

Any other value for CUSTNO causes the **CONTINUE** to execute and the **CALL** to the procedure LAST is not executed.

Example 2:

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

A value of CUSTNO less than 10000 causes the **CALL** command to procedure FIRST to execute. The **CONTINUE** bypasses the **CALL** command to procedure SECOND.

A value of CUSTNO greater than or equal to 10000 causes the **CALL** command to procedure SECOND to execute.

DO Command

```
DO {[WHILE logical-expression]
    [UNTIL logical-expression]
    [FORALL segment-name]
    [FOR integer] |
    [CASE]}
```

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, as discussed in Section 1, 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 which is evaluated at the beginning of each iteration of the explicit loop (i.e., at the **DO** command). If the condition is true, another iteration of the explicit loop is performed; otherwise, the explicit loop is terminated and control is transferred to the statement after the **END** command associated with the **DO** block. Therefore, the conditions on the logical expression must be changed within the **DO** block in order to terminate the explicit loop. This keyword phrase cannot be used with the **CASE** keyword.

UNTIL *logical-expression*

Specifies a logical expression which is evaluated at the end of each iteration of the explicit loop (i.e., at the **END** command). If the logical expression is true, the explicit loop is terminated and control is transferred to the statement after the **END** command associated with the **DO** block. Otherwise, another iteration of the explicit loop is performed. Therefore, the conditions on the logical expression must be changed within the **DO** block in order to terminate the explicit loop. This keyword phrase cannot be used with the **CASE** keyword.

FORALL *segment-name*

Specifies the name of a non-root segment. The **DO** block is executed for all of the segment occurrences. Each iteration of the **DO** block processes the next occurrence of the specified segment. This keyword phrase cannot be used with the **CASE** keyword.

FOR *integer*

Specifies an integer value from 1-99999 that represents the maximum number of times the **DO** block is executed. 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, then the **DO** block is automatically terminated at the maximum number. The **FOR** keyword phrase cannot be used with the **CASE** keyword.

CASE

Specifies that **CASE** command blocks follow. This keyword has no operands. A **DO CASE** block can optionally have an **ELSE** command within it. The first **CASE** command whose conditions are met is executed within the **DO CASE** block and then control is passed 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, as discussed in Section 1, still occurs. The **CASE** keyword cannot be used with the **WHILE**, **UNTIL**, **FORALL**, and **FOR** keywords.

If the condition on the **WHILE** operand is initially false, the **DO** block is *not* executed. Conversely, the **UNTIL** keyword makes it possible for statements within the **DO** block to be executed at least once even though the condition is initially true.

The **WHILE**, **UNTIL**, **FORALL**, and **FOR** keywords can be used in combination with each other with the exception that **FORALL** and **FOR** are mutually exclusive.

If the **FORALL** and **WHILE** keywords are both specified, the first occurrence of the segment is retrieved before the **WHILE** condition is evaluated.

Example 1:

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

Since the field T.COUNT is set to zero before starting the **DO** block, the condition on the **DO** command is true and the procedure statements within the **DO** block are executed. In the second interaction of the explicit loop, the condition on the **DO** command fails and the procedure statements are bypassed. The next procedure statement after the **END** for the **DO** block is processed. The **CALL** command within the **DO** block is only processed once each time the procedure is executed.

Example 2:

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

Since T.RECNUM is set to zero before starting the **DO** block, the procedure statements within the **DO** block are executed. 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. This time at the end, T.RECNUM contains a 2 and the **UNTIL** condition fails again, allowing the **DO** block to be executed a third time. At the end of the **DO** block this 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.ORDITEM
  LET T.FIELD1 = 1.FIELD1 + T.FIELD1
END
```

The segment 1.ORDITEM is a child segment in a data base. If there are no ORDITEM segments within the parent segment, the **DO** fails and the procedure statements in the **DO** block are bypassed.

The **DO** block is executed once for each occurrence of an ORDITEM segment. The field T.FIELD1, therefore, accumulates a total of all of the values of 1.FIELD1 from each synchronized record of file 1. The **DO** block is terminated at the end of file on file 1 and the statement following the **END** command is then executed.

Example 4:

```
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 first **CASE** command is evaluated and the logical expression is false. The second **CASE** command is evaluated and the logical expression is true. The procedure statement following the second **CASE** command is executed and control is passed to the **END** command corresponding to the current **DO CASE** block.

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

The procedure statements following the first **CASE** command evaluated with a true logical expression are executed and control is immediately passed to the **END** command for the **DO CASE** block.

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, then the T.TITLE field is assigned the portion of the state corresponding to the area code. If the area code is not one of the Texas state area codes, the **ELSE** command is executed and the T.TITLE field is set to OUTSIDE OF TEXAS.

END Command

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

Example 1:

```
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
```

An **END** command is required 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]
           [EDLEN edit-length]
           [INIT initial-value]
```

The **FIELD** command defines a temporary field within a procedure. The temporary field, once defined on the **FIELD** command, can be used in other procedures. It is a good idea 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. Field names can be eight characters long and must start with an alphabetic character. See Chapter 1, “Terminology, Syntax, and Processing” for valid field names. A colon is placed 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. All FIELD statements must be at the beginning of a procedure
TYPE field-type	Specifies the field type. This operand is a one-character code corresponding to the storage type of the field being defined (e.g., C for character type field, Z for zoned decimal). Valid entries can be found in Appendix B, “Technical Notes”
LENGTH field-length	
LEN	Specifies the field length. This operand must be an integer constant with a value within the range of the limits imposed by the field type. Default field lengths for each field type can be found in Appendix B, “Technical Notes” .
DECIMALS decimal-places	
DEC	Specifies the number of decimal places for numeric fields. This operand must be an integer between 0 and 9 and cannot be larger than the length of the numeric field.
EDLEN edit-length	Specifies an overriding edit length to be use to display this field. If you omit this operand, the maximum number of positions required to display this field is assumed
INIT initial-value	Specifies an initial value. This operand is a constant whose type corresponds to the field type (e.g., for numeric type fields use numeric digits and, if needed, a negative sign and decimal point).

Example:

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

The temporary field established can be referenced by using the qualifier T in front of the field name (e.g., T.FRSTTIME). The field contains character type values (A-Z, 0-9, blanks, etc.) and is one character long with an initial value of Y.

GO Command

GO [TO] *jump-to-label*

The **GO** command is a branch forward statement to another labeled procedure statement in the current procedure. Labels on lines are created 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. This operand, *jump-to-label*, must be a label on a subsequent statement in the current procedure. The colon used on the labeled procedure statement is not used on the **GO TO** command.

The **GO** may only branch forward to a subsequent labeled statement; it may not branch backward to a prior labeled statement. The labeled statement must be in the current procedure.

A **GO TO** statement can be used to jump out of an **IF-END** or **DO-END** group of statements. If used in this manner, it overrides the **WHILE**, **UNTIL**, **FOR**, and **FORALL** conditions governing the **DO** loop

Example 1:

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

The periods represent procedure code not shown because it is not important to the example of the **GO** command. The example shows the syntax of the **GO** branching to a labeled line forward in the procedure called NEXT. 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 NEXT are bypassed and processing continues with the **LET** command on the labeled line.

Example 2:

```
IF ORDRDATE EQ P.MMDYY
  CALL PROCEDURE ORDER
ELSE
  LET T.ERROR = ORDRDATE
  LET T.MSG = 'INVALID ORDER DATE'
  GO TO ERROR
END
;
IF ITEMNO EQ P.NUMERIC
  CALL PROCEDURE ITEM
ELSE
  LET T.ERROR = ITEMNO
  LET T.MSG = 'INVALID ITEM NUMBER'
  GO TO ERROR
```

```

END
.
.
.
.
ERROR: CALL PROCEDURE ERROR

```

The label **ERROR** is on a **CALL** command for an error procedure. Only the first error detected is processed by this code. The periods between the **END** command and the **ERROR** labeled statement mean that some of the procedure has been omitted from the example.

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 procedure **ORDER** is executed, then processing continues with the next **IF** command. If the field **ITEMNO** does not contain all numerics according to the numeric 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]
```

The **IF** command begins an **IF** block that contains a group of statements that are to be performed when the logical expression on the **IF** command is true. Control is transferred to the corresponding **ELSE** or **END** statement for this **IF** block when the logical expression is false. An **END** command delimits 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) and include the **PF** function.

THEN Is an optional entry with no operands. It is provided for clarity and readability where desired..

Example 1:

```

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

```

When the field **CUSTNO** contains 00002, the logical expression on the **IF** command is evaluated as false, control is passed to the **ELSE** command within the **IF** block, and the field **T.SET** is set to **OFF**.

Example 2:

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

If the field ORDRDATE contains 010192 and the field F.TODAY contains 022493, 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 system field TODAY minus 1 is 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.

LEAVE Command

```
LEAVE
```

The **LEAVE** command exits the corresponding **DO** block immediately (i.e., 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.

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 = 50
  CALL MODULE GETAMT USING 1.AMOUNT
  IF 1.AMOUNT GE 0
    LET T.TOTAL = T.TOTAL + 1.AMOUNT
    LET T.COUNT = T.COUNT + 1
  ELSE
    LEAVE
  END
END
```

In Example 1 the **LEAVE** command transfers control out of the **DO** block if the module GETAMT returns a negative amount before 50 amounts are processed. If a negative amount is returned before the **DO UNTIL** is satisfied, the false condition on 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 PROCEDURE ERROR
```

```

        LEAVE
      END
    END

```

This section of code counts all of the completed orders. If any order is not complete (i.e., ORDCMPLT is not Y), the **ELSE** command is executed along with the subsequent procedure statements. The procedure ERROR is processed and then the **DO** block is exited 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 control is passed 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
```

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

FIELD *result-field* Specifies the receiving field. The operand must be a field name or PF function.

source-expression Specifies the "from" field or arithmetic expression. This operand must be a name, constant, or arithmetic expression which may include a PF function..

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 hand side of the equal sign.

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

Example 2:

```
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 3:

```
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'

MSG Command

```
MSG [TEXT] `text of message' | field-name [NOROLLBACK]
```

The **MSG** command is used to terminate the interaction and display the specified message when screen field validation fails.

<i>`text of message'</i>	Specifies the text of the message to be displayed. Maximum is 31 characters..
<i>field-name</i>	Specifies the field to be displayed..
NOROLLBACK NOROLLB	Specifies that data base updates are not to be backed out.

Example 1:

```
IF F.ATTNID NE 'PF010'
  MSG TEXT 'YOU MAY ONLY PRESS PF10' NOROLLB
END
```

In this example, if any key other than PF010 is pressed, the message YOU MAY ONLY PRESS PF10 is displayed and data base updates are not backed out.

Example 2:

```

      IF S.ACTION EQ 'X'
        LET T.COUNT = T.COUNT + 1
        IF T.COUNT > 1
          MSG 'YOU CAN ONLY ENTER ONE 'X' PER SCREEN'
        END
      END

```

This procedure makes sure that only one X is entered on the screen. If more than one X is entered, the message YOU CAN ONLY ENTER ONE `X' PER SCREEN is displayed.

RETURN Command

```

      RETURN

```

The **RETURN** command is used 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 since the end of a procedure constitutes an automatic **RETURN**.

In a subroutine procedure, control is automatically passed back to the calling procedure after all of the statements have been processed.

Example 1:

```

      IF ORDCMPLT = 'Y'
        RETURN
      END
    ;
      IF DATE LE '930101'
        RETURN
      END
      CALL PROCEDURE ALLOTHRS

```

These procedure statements make up a subroutine procedure which can be called from any other procedure. The **RETURN** commands, when executed, pass control back to the calling procedure. If ORDCMPLT is equal to Y or the DATE is less than or equal to 930101, the subroutine procedure is exited immediately by one of the **RETURN** commands. The subroutine procedure is automatically exited after processing the call to procedure ALLOTHRS; the **RETURN** is implied.

SET Command

```
SET [ERROR] input-screen-field
```

The **SET** command is used to specify the name of the input screen field in error that is to be processed by error handling. Each time this command is executed, the value in the system field ERROR is incremented by one.

input-screen-field Specifies the name of the screen field to be processed by error handling. The default error handling for a screen field is highlighting the field. This can be overridden in the screen definition when defining the screen field.

Example:

```
IF S.FIELD1 NE P.INTEGER
  SET ERROR S.FIELD1
END
IF S.FIELD2 GT 2500
  SET ERROR S.FIELD2
END
IF F.ERROR GT 0
  MSG 'FIX THE INCORRECT FIELDS(S) '
END
```

In this example, two screen fields are being validated. If screen field FIELD1 is not an integer, it is set in error by the **SET** command. Similarly, FIELD2 is tested for a value greater than 2500. Then, the system field ERROR is checked for a value greater than zero.

If all fields passed the validation, the value in ERROR would be zero and the procedure would be executed with no error processing. However, if either of the fields failed validation, the value in system field ERROR would be greater than zero, error handling would be performed (which normally causes the screen field to be highlighted), and the message FIX THE INCORRECT FIELD(S) would be displayed.

TRACE Command

```
TRACE [FIELD] {`constant' | field-name} [IN {CHAR | HEX}]
```

The **TRACE** command is used to display literals or fields during application execution.

Note: **TRACE** commands are always valid within procedures; however, they are only executed if Procedure Tracing has been enabled.

'constant'	Specifies a character constant to be displayed in the trace output. The constant must be a character constant delimited by single quotation marks; numeric constants are not allowed. The maximum length is 31 characters.
field-name	Specifies the field to be displayed in the trace output
IN CHAR	Displays the constant or field in character format. This is the default for constants when the IN clause is omitted. If IN CHAR is specified for a numeric field, it is changed to IN HEX during application execution
IN HEX	Displays the constant or field in both character and hexadecimal format. This is the default for fields when the IN clause is omitted. IN HEX is always used to display numeric fields even if IN CHAR is specified

Example:

```
TRACE A IN HEX
TRACE 'PRIOR TO CALCULATION' IN CHAR
TRACE T.AMOUNT IN HEX
LET T.AMOUNT = A * 100
TRACE 'AFTER CALCULATION'
TRACE T.AMOUNT
```

The trace data output from the above example is:

- Contents of field A in both character and hexadecimal format.
- Literal PRIOR TO CALCULATION displayed in character format.
- Contents of field T.AMOUNT in character and hexadecimal format.
- Literal AFTER CALCULATION displayed in character format (default for literals).
- Contents of field T.AMOUNT displayed in character and hexadecimal format (default for fields).

TRANSFER Command

```
TRANSFER [TO] { NEXT_FUNCTION | SCREEN_CONTROL | OUTPUT }
```

The **TRANSFER** command is used to exit procedures immediately and give processing control to another type of procedure.

NEXT_FUNCTION Exits this procedure and transfers control to the next automatic function. If executed in a Validate procedure (type 1), it skips to automatic input data validation. If executed in a Retrieve procedure (type 2), it skips to in-memory updates. In an Update procedure (type M), it skips to update the data bases. In an Output procedure (type O), it stops building the output screen and displays the output buffer.

SCREEN_CONTROL Exits this procedure, skips all other input control procedures of this type and the automatic functions, and proceeds to screen control.

OUTPUT Branches to output screen processing. Valid only in Output procedures.

Example:

```
IF CUSTOMER LT '01000'  
  TRANSFER TO SCREEN_CONTROL  
END  
.  
.
```

Only CUSTOMER values not less than 01000 are processed through the remainder of the procedure after the **END** command. For CUSTOMER values less than 01000, the **TRANSFER** command is executed.

ASL Examples

Following are some examples of ASL usage showing the use of the language in the context of a procedure within an application. The purpose of these examples is to illustrate how the language is used to perform a variety of operations which use the various features of the language.

Example 1:

This example is of a simple procedure which converts feet to centimeters. The value in field DISTANCE is converted to centimeters and the result is stored back into the field DISTANCE in the logical record. There is no change to the physical data unless an update was requested.

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

Because of the set operation of ASL, this one-line procedure transforms all instances of the field DISTANCE in the data base. When the field DISTANCE is subsequently displayed, it has already been transformed into units of centimeters by this procedure.

Example 2:

This example shows the use of the partial field function to build a key for a log file. The log file key has the format YY/MM/DDHH.MM.SS, where YY/MM/DD is the current date and HH.MM.SS is the current time. The key is built in a temporary field.

```
;Build key for log file.  
;  
T.LOGKEY: FIELD TYPE C LENGTH 16  
LET PF(T.LOGKEY 1 8) = ' / / '  
LET PF(T.LOGKEY 1 2) = PF(F.TODAY 5 2)  
LET PF(T.LOGKEY 4 2) = PF(F.TODAY 1 2)  
LET PF(T.LOGKEY 7 2) = PF(F.TODAY 3 2)  
LET PF(T.LOGKEY 9 8) = F.TIME
```

Note: F.TODAY is a system field containing the current date in the format MMDDYY.

Example 3:

This example is of a filter which does not allow the viewing of any employee records for employees whose salaries are \$50,000 or greater.

```
;Bypass display of employees whose salaries are $50,000 or greater.
;
IF SALARY = 50000
  CONTINUE
ELSE
  TRANSFER TO OUTPUT
END
```

With this procedure as a type O procedure for the screen in which the Employee file is browsed, all records for employees whose salaries are \$50,000 or greater are not displayed on the output screen.

Example 4:

This example illustrates simple validation and error handling for an input screen as well as transferring to specific points within the interaction. This procedure is executed as a type 1 procedure.

```
;Validate function key and verify employee number entered.
;
DO CASE
CASE WHERE F.ATTNID = 'PF001'
  TRANSFER TO SCREEN CONTROL
CASE WHERE F.ATTNID = 'PF002'
  TRANSFER TO NEXT FUNCTION
CASE WHERE F.ATTNID = 'ENTER'
  IF S.EMPNO = ' '
    SET ERROR S.EMPNO
    LET F.MSGLINE = 'EMPLOYEE NUMBER IS REQUIRED.'
    MSG F.MSGLINE
  END
ELSE
  LET F.MSGLINE = ' '
  LET PF(F.MSGLINE 1 30) = 'INVALID KEY PRESSED. PLEASE PR'
  LET PF(F.MSGLINE 31 30) = 'ESS PF001, PF002, OR ENTER.'
  MSG F.MSGLINE
END
```

This procedure validates the program function key pressed on an input screen. If the program function key is not valid, the MSG command is used to re-display the input screen with the error message "INVALID KEY PRESSED. PLEASE PRESS PF001, PF002, OR ENTER."

If PF001 is pressed, no input processing is required for the screen, so a transfer to screen control is performed where a menu screen is selected for output. If PF002 is pressed, input processing for the screen is required, however, all remaining type 1 procedures are to be skipped (NEXT FUNCTION).

When the Enter key is pressed, an employee number must be provided on the input screen. If an employee number is not entered on the screen (S.EMPNO), the SET command causes the field to be highlighted and the MSG command results in the input screen being re-displayed with the message "EMPLOYEE NUMBER IS REQUIRED."

Relationship of ASL Statements to VISION:Transact

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

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

Function	VISION:Transact Operation
PF	A partial field operation is performed on the field named in the function. The PF function can be used on as many fields in the command as needed
Command	VISION:Transact Operation
CALL	Branches to a subroutine request which has more PR statements. An external program can also be called with this command.
CASE	Creates a logical expression or a compound logical expression.
CONTINUE	Performs a GO END.
DO	Creates a loop depending on the keyword phrases used on the command. The keywords UNTIL and WHERE set up back branch loops with the FOR 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 series of conditional PRs with NS branching logic.

Command	VISION:Transact Operation
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 where the ELSE is optional within an IF command.
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 coded before the 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 put into a subroutine internally so that the LEAVE in a DO block performs a GO RETURN.
LET	Creates an R (replace) operation.
MSG	MS statement to terminate interaction and display message text.
RETURN	Sets up a GO RETURN from a subroutine procedure.
SET	SE operator to increment the count in system field ERROR and identify screen field to be processed with error handling.
TRACE	Creates a ?C operator to display a constant or field in character format or a ?X operator to display a constant or field in both character and hexadecimal format.
TRANSFER	A GO OUTPUT, GO NEXT FUNCTION, or GO SCREEN CONTROL.

Technical Notes

This appendix contains additional information about topics described in [Chapter 1, “Terminology, Syntax, and Processing”](#), [Chapter 2, “Built-In Function”](#), and [Chapter 3, “Procedure Statements”](#). 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).

The information is organized by topic. The following topics are incorporated into this appendix:

- Reserved words.
- Qualifiers
- Patterns.
- Valid field types and default field lengths.

Reserved Words

Field names should not duplicate system defined names(reserved words) such as commands, functions, keywords, or operators because they may cause misleading interpretation of the statement. The following is a list of restricted ASL words. Conflict with these reserved words can be avoided if the name is surrounded by double quotation marks or all keywords are entered in the statement.

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

LU	PS	SSCOUNT
MAX	PSTART	START
MIN	R	STR
MISSPASS	RATIO	STRAN
MN	RD	STRING
MNUMBER	RE	SUB
MOD	READ	SUBFILE
MODE	REL	TAB
MODULE	RELEASE	TABLE
MS	REP	TB
MSG	REQ	TERMID
MSGLINE	REQUEST	TEXT
MSTART	REPLACE	THEN
MSTATUS	REPORT	TI
NE	RESTART	TIME
NEAREST	RET	TL
NEXT	RETURN	TN
NEXT_PROC	RETURNCD	TO
NEXTPROC	RG	TODAY
NOTEQUAL	RN	TOT
NRST	RNUMBER	TOTAL
NS	ROUNDING	TRACE
OPERID	ROW	TRAIL
OPERL	RS	TRAN
OUTPUT	RSTART	TRANCODE
OWN	RSTATUS	TRANSFER
PAGE	RTO	TRL
PASSWORD	SCAN	TS
PAT	SEG	TYPE
PATTERN	SEGNAME	UNTIL
PCT	SET	USING
PF	SL	VAL
PLI	SMALLER	VALIDATE
PN	SMLR	WHERE
PNUMBER	SN	WHILE
PROC	SR	WITH
PROCEDURE	SS	XTRAN

Qualifiers

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

Qualifier	Type or Location
Blank or N	Synchronized or pseudo file
1-9	Synchronized or pseudo files 1-9
T	Temporary field
F	Flag field
W	Working storage field
V	PCB mask field
P	Date or numeric pattern
S	Input screen field

PATTERNS

Pattern names can be used in the logical expression (with operators EQ or NE) on the **IF** or **CASE** statements. A pattern name must be preceded by the letter P and a period. Pattern names and their meanings are as follows:

Patterns	Meaning
MMDDYY MMYYDD DDMMYY DDYYMM YYDDMM YYMMDD	These patterns describe the various six-character date formats. All of these require six left-justified numeric digits. This validation will assure that year (YY) is numeric, month (MM) is numeric and in the range 01 through 12, and day (DD) is numeric and is valid within month (includes validation for leap years).
TODAY	This pattern corresponds to the format of the TODAY flag field. It will be the same as one of the formats above. TODAY flag format can be specified in M5PARM and this can be installation specific.
JULIAN	This pattern describes a date in the format YYDDD, where year (YY) must be numeric and day (DDD) must be in the range 001-365 (001-366 for leap years).

Patterns	Meaning
NUMERIC	This pattern describes a field that, except for leading and trailing spaces, contains at most one sign (a minus, ``-"), numeric digits, grouping characters, and at most one decimal point (.). If the field contains a sign, then it must immediately precede the first numeric digit or the decimal point. If field has a decimal point, it must be followed by at least one numeric digit. A blank field is considered to be numeric. The field may contain grouping characters to the left of the decimal (for example, a comma); however, the grouping characters must separate groups of three digits (except for the leftmost group which may contain one or two digits).
INTEGER	This pattern describes a field that, except for leading and trailing spaces, contains at most one sign (a minus, ``-"), numeric digits, and grouping characters. If the field contains a sign, then it must immediately precede the first numeric digit and there must be at least one digit. The field may not contain a decimal point. A blank field is considered to be an integer. The field may contain grouping characters (for example, a comma); however, the grouping character must separate groups of three digits (except for the leftmost group which may contain one or two digits).
FLOAT	This pattern describes a number in floating point (scientific) notation. A field is considered to be in floating point format if it is a decimal number followed by an exponent. An exponent is the character E followed by a + or -, and a one or two digit number. The sign may be omitted, in which case a plus, +, is assumed.

Valid Entries for Field Type and Length

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	1-8 Bytes	8

Field Type	Field Range	Default Field Length
E Floating point binary	4 Bytes only	4
F Fixed point binary	1-4 Bytes	4

This appendix describes the available VISION:Transact flag fields.

VISION:Transact provides flags or internal indicators which you can interrogate or set to indicate the existence of certain conditions. Flag fields are identified by the qualifier F. Their general uses are shown in the following table. Flags are not re-initialized during a recycle interaction, except for the date and time flag fields.

Note: * = International Standard Organization (ISO) date format.

Flag Name	Application
TODAY DATE ISDATE* JULIAN	Provides the operating system date (in one of four possible formats) to the user.
TIME	Provides the time of day that execution was started.
CSTATUS RETURNCD	Provides status information regarding CALL statements that invoke external user-written modules.
ATTNID	Provides information about the terminal program function keys that caused transmission of the input message.
ERROR	Provides an automatic count of the number of times the SET command is executed.
TRANCODE	Provides the name of the current transaction that is processing the input screen data.
GOSC	Provides the type of the procedure (1, 2, or M) from which a TRANSFER TO SCREEN_CONTROL branch was issued.
TERMID USERID OPERID OPERCL (CICS only)	Provides information about the terminal or operator.

Flag Name	Application
MSGLINE	Enables you to explicitly place the error message line on the screen, and to place user data in the error message line on non-error displays.
PSTART or PS PNUMBER or PN RSTART or RS RNUMBER or RN MSTART or MS MNUMBER or MN LSTART or LS LNUMBER or LN	Provides numeric fields to be used as counters or indexes (useful during back branch operations). Abbreviations (2 characters) are allowed in the partial fielding start and length operands of the PF function.
HIGHVALU LOWVALU	Provides high value or low value character strings useful in the testing and setting of high or low values.
PAGENO	Used to display page numbers for output screen pages.
SCRNAMEI SCRNAMEO	Provides the name of the input and output screen being processed in the interaction.
TCONTROL TAPPFLOW TPROC TSEGACTION TILOOP TDLOOP TSSDBIO TDSDBIO	Provides the ability to enable or disable all or individual execution tracing options if the application was generated to include tracing

When specifying a flag name, use F as the qualifier.

TODAY, DATE, ISDATE, and JULIAN Flags

All four flags produce the same date but in the following different formats:

TODAY	Six characters as follows: MMDDYY (where MM = month, DD = day, YY = year), e.g., 101493. This format can be controlled by M5PARM. The format shown is the default.
DATE	Twelve characters as follows. Three alphabetic characters for month, space, two numerals for day, comma, space, and four numerals for year, e.g., OCT 14, 1993. This format can be controlled by M5PARM. The format shown is the default.
ISDATE*	Eight characters as follows: YYYYMMDD (where YYYY = year, MM = month, DD = day), e.g., 19931014. Note: * = International Standard Organization (ISO) date format.
JULIAN	Five characters as follows. YYDDD (where YY = year, DDD = day), e.g., 93288 The formats for the TODAY and DATE flags are influenced by specifications in M5PARM (refer to the Installation Procedures).

TIME Flag

The TIME flag contains the time of day (on a 24-four hour clock) when application execution begins. The data is stored as an eight-byte character string and takes the form:

HH.MM.SS

where:

HH = hours

MM = minutes

SS = seconds

Single digits will contain a leading zero. For example, six minutes and four seconds after 9:00 A.M. will be represented as 09.06.04, and 12 seconds and 46 minutes after 4:00 P.M. will be represented as 16.42.12.

CSTATUS and Returned Flags

The status of the call to an external user-written module (using either keyword, CALL MODULE or CALL DLI) in IMS TM applications is determined from two flag specifications: RETURNCD and CSTATUS. In CICS applications, you can use only the CSTATUS flag.

Call status information is obtained from the Call Status (CSTATUS) flag for IMS TM and CICS. The CSTATUS provides a method of determining in a VISION:Transact procedure if a call was suppressed by VISION:Transact. A call will be suppressed if a parameter specifies an invalid or missing field.

The CSTATUS is a four-byte character string flag field which may have the following values:

PMIS	Parameter data field missing
PINV	Parameter data field invalid
(blank)	Call successful

The CSTATUS flag should be examined immediately after a call to an external user-written module, since a suppressed call would not allow the called module to be executed.

The RETURNCD flag is a VISION:Transact flag that shows the result of the call as set by the called routine (IMS TM only).

The VISION:Transact Return Code flag field (RETURNCD) values are set by the called routine, while the Call Status flag field (CSTATUS) values are set by VISION:Transact. Following any call, the contents of RETURNCD may be examined (after CSTATUS flag) to determine the results of the call as set by the called routine. The value in the RETURNCD field is the value in General Register 15 upon return to VISION:Transact from the called routine. This value may be set by an Assembler program by simply loading the register with the desired value; COBOL programs use the RETURN-CODE keyword.

The RETURNCD flag is a four-byte, fixed point field. However, VISION:Transact edits it as though it were a two-byte fixed point field, so its display width is seven and the maximum printable value is 99,999.

When testing for a specific value in the RETURNCD, the developer must know the possible settings, which depends on the called routine.

ATTNID Flag

The ATTNID flag allows you to determine which terminal key was used on the last input.

The ATTNID flag is a five-byte character field which may contain one of the following values:

ENTER	Operator pressed the Enter key.
PFnnn	Operator pressed PF key nnn where nnn can contain a value from 001 to 036.
(CICS) CLEAR	Operator pressed the Clear key. Clear is only passed to a VISION:Transact application if the CICSKEY handling entry in M5PARM or the RC statement indicates that the Clear key is to be processed.
(CICS) PANnn	Operator pressed PA key nnn where nnn is a value of 001 to 003. PA keys are only passed to a VISION:Transact generated application if the CICSKEY handling entry in M5PARM or the RC statement indicates PA keys are to be processed.

Note: If CICS BMS paging commands have been assigned to PA or PF keys, and multiple pages of a screen defined with Output Control entry of A in the screen definition or OC OUTLIM in screen control have been output, then the PF or PA keys will not be passed to the applications while the operator is paging, but are intercepted by BMS.

Note: Since the ATTNID flag is a character field, PF or PA key numbers must always be specified with leading zeroes (e.g., PF1 is not the same as PF001).

ERROR Flag

The ERROR flag is used in conjunction with the SET command in type 1, 2, and M procedures. The ERROR flag is a two-byte fixed point field which is initialized to zero at the start of each interaction. It is automatically incremented by one each time the SET command is executed.

The value of this flag can be interrogated to determine whether input screen fields are in error. It can also be displayed in the text of a MSG command. This technique provides a means for indicating the number of screen fields with errors.

TRANCODE Flag

The TRANCODE flag is an eight-byte field. On entry to a VISION:Transact generated application, the current transaction name is placed in this flag field. For IMS TM conversational transactions, the TRANCODE flag is initialized to the transaction code contained in the SPA.

You can manipulate this flag by permitting the user to enter a new value on the screen or by supplying one yourself in procedure logic. The transaction specified in the flag processes the next screen's input. The contents of this flag may be modified by the application at any time prior to the building of the output screen.

The TRANCODE flag facilitates the sharing of screens between applications, because you are not required to specify in the screen processing definition a specific transaction to process a specific screen. The value in the field determines the transaction that will process the screen entered. Thus, a screen can be input to various transactions.

The use of the flag in an IMS TM application requires that it appear on the screen and have an S0 statement associated with it. If you want, the field can be specified as dark and protected to guard against user error and allow only you to update the flag in a procedure. Note that the flag field must not appear in a multiply occurring screen segment and must have a B (both) Input/Output specification. The value in the field is used to determine the transaction that will process the output screen on subsequent input processing.

To use the flag in a CICS application you need not place the TRANCODE flag on the screen. Instead, you can simply update the flag in a procedure. To update the flag, place the name of the next transaction in it (CICS transaction name is limited to four characters). This update can be done any time before the application completes building the output screen. The VISION:Transact generated application will use the flag as the TRANSID upon RETURN to CICS. See the *Advantage VISION:Transact Environment Guide* for more information on using the TRANCODE flag.

GOSC Flag

The GOSC flag is a one-byte, character field that is used together with the procedure command TRANSFER TO SCREEN_CONTROL. When a type 1, 2, or M procedure executes a TRANSFER TO SCREEN_CONTROL branch, then this field will contain the type of the procedure that executed the branch. If no such branch is issued, then GOSC contains a blank. The only possible values for GOSC are 1, 2, M, or blank.

TERMID, USERID, OPERID, and OPERCL Flags

These flag fields contain the identification of the terminal, the user or operator who initiated the transaction, and, for CICS applications, the operator class.

TERMID is an eight-byte character field. In an IMS TM application, TERMID is initialized to the LTERM name in the I/O PCB. In a CICS application, it is initialized to the value of EIBTRMID in the Exec Interface Block. Only the leftmost four bytes of TERMID are significant in CICS. The rightmost four bytes contain blanks.

The USERID flag is an eight-byte character string field. Under CICS, the flag contains the user identifier of the person who is signed on; it contains blanks if the CICS release is prior to 1.7 or if the VISION:Transact task is not initiated from a terminal. Under IMS TM, the USERID flag contains the user identification field from the I/O PCB; this information is also available through the OPERID flag.

OPERID is an eight-byte character field. In an IMS TM application, OPERID is initialized to the USERID in the I/O PCB. In a CICS application, the leftmost three bytes of OPERID are initialized to the value returned by the CICS ASSIGN OPID command. Bytes four through eight contain blanks.

OPERCL is a three-byte character field. In a CICS application, OPERCL is initialized to the value returned by the CICS ASSIGN OPCLASS command. For IMS TM applications, this flag field contains blanks.

MSGLINE Flag

When your application encounters an error condition, it re-displays the screen that was input to the application with an error message. This error message by default is placed on the bottom line of the terminal.

The MSGLINE flag is 255-byte, character field and it is initialized to blanks at the start of each interaction. It allows you to place the line on which the error message will be displayed anywhere in the root screen segment. In addition, because the contents of the MSGLINE flag are displayed during normal output processing, you can place data to be displayed into the MSGLINE flag field.

If you do not reference the MSGLINE field in the Screen Contents window (or S0 statement), then the message line is automatically placed on the last line of the screen. To place the message somewhere other than the bottom line of the screen, you first enter a series of at least 79 screen field designators when painting the screen (or SL statement). This screen field defines the location of the error message when the input screen is re-displayed. You then specify the MSGLINE flag field in the Contents window (or S0 statement).

To output data into the error message line, move whatever data you want into the flag MSGLINE in a procedure. That data will be displayed in the error message line when the screen is output as long as the MSGLINE flag is an output field. This data will not be displayed during an error cycle unless flag MSGLINE was named as an operand of the MSG command.

Dynamic Partial Field Flags

There are eight flag fields that can be used in partial field operations. They are PSTART, PNUMBER, LSTART, LNUMBER, MSTART, MNUMBER, RSTART, and RNUMBER. The field names can be abbreviated to PS, PN, LS, LN, MS, MN, RS, and RN, respectively. These fields are four-byte, fixed point, integer fields that are initialized to zero at the start of an interaction.

The two-character abbreviations are used in the PF function start and length operands.

For example, if the value of LS is 10 and value of RN is 3, and LS is used in the start operand and RN is used in the length operand of the PF function, then positions 10, 11, and 12 of the referenced field will participate in the operation. Note that this is equivalent to using a 10 in the start operand and a 3 in length operand.

When you use these dynamic partial field flags, you must be aware of the rules that apply to partial fielding:

- Partial field start must be between 1 and the length of the field.
- Partial field length must be greater than zero.
- Partial field start plus partial field length must not exceed the length of the field.
- Neither partial field start nor partial field length may be invalid.
- If these rules are violated, then one of the following occurs:
- If the field being partial fielded is a source field for some operation, then the field is treated as an invalid field.
- If the field being partial fielded is a result field, then the entire field becomes invalid.

HIGHVALU and LOWVALU Flags

The HIGHVALU flag is an eight-byte character field that contains high values (X'FF's). The LOWVALU flag is an eight-byte character field that contains low values (X'00's). These flags can be useful in the testing and setting of high or low values in other fields.

PAGENO Flag

The PAGENO flag is a two-byte fixed point binary field that contains the page number of the output screen. When page numbers are desired on output screen pages, a screen field must be defined on the screen layout and the corresponding S0 statement for the screen field must name flag PAGENO as the source field for output.

If flag PAGENO is interrogated during output processing, it is important to understand that the contents of the flag reflects the page number of the previous TRANSFER TO OUTPUT executed by the screen's output procedure. The flag contains a 1 on entry to a screen's output procedure and is incremented by 1 *after* a screen page is output by VISION:Transact.

SCRNAMEI and SCRNAMEO Flags

These flags are eight-byte character string fields.

The SCRNAMEI flag contains the name of the screen that was input to the interaction. During a recycle interaction (the output screen is recycled as input), SCRNAMEI contains the name of the recycled screen upon input. Flag SCRNAMEI is initialized at the start of each interaction cycle to the name of the input screen; this includes recycle and CICS conversational interactions.

The SCRNAMEO flag contains the name of the screen currently being output in the interaction. This flag contains blanks during input processing (type 1, 2, or M procedures or subroutine procedures called from these procedures). During output processing, this flag contains the name of the output screen currently being processed. Flag SCRNAMEO is initialized at the start of each interaction cycle to blanks; this includes recycle and CICS conversational interactions. The flag is then re-initialized during screen control for each output screen selected. The name of the selected output screen is placed in the flag prior to execution of the output procedure and building of the output screen.

Execution Tracing Flags

There are eight flag fields that can be used to enable or disable all or individual execution tracing options, provided an application has been generated to include tracing. These trace flags which correspond to the entries on the Environment window (or ?T statement), are one-byte character fields, and have the following meanings and allowable settings:

Flag Name	Purpose	Description
TCONTROL	Trace Control	0 indicates that all tracing is disabled, 1 indicates tracing is enabled and that individual traces are enabled or disabled, based on the contents of their associated trace flag.
TAPPFLOW	Application Flow Tracing	0 indicates disabled, 1 indicates level 1 tracing is enabled, and 2 indicates level 2 tracing is enabled. Level 1 outputs all level 1 application flow messages (AFnn) and teleprocessing interface calls for screens and saved working storage/pseudo files (TPnn messages). Level 2 outputs all level 1 and 2 application flow messages and all TP interface calls and I/O areas for screens and saved working storage/pseudo files.
TPROC	Procedure Tracing	0 indicates disabled and 1 indicates enabled which also enables trace commands/operators.
TSEGACT	Segment Action Tracing	0 indicates disabled and 1 indicates enabled.
TILOOP	Intermediate Structure Loop Tracing	0 indicates disabled and 1 indicates enabled.
TDLOOP	DSDB Loop Tracing	0 indicates disabled and 1 indicates enabled.
TSSDBIO	SSDB I/O Call Tracing	0 indicates disabled, 1 indicates level 1 enabled (I/O areas are not displayed), and 2 indicates level 2 enabled (I/O areas are displayed).
TDSD BIO	DSDB I/O Call Tracing Level	0 indicates disabled, 1 indicates level 1 enabled (I/O areas are not displayed), and 2 indicates level 2 enabled (I/O areas are displayed).

The TCONTROL flag is initialized to 1 at the beginning of application execution if the Environment window specifies ENABLE ANY (or the ?T statement contains an entry of 1) in Trace Control; otherwise it is initialized to 0. The remaining flags are initialized at the beginning of application execution to the value of their corresponding Environment window or ?T statement execution tracing entry. Note that a value of blank is treated as 0 in the corresponding flag field.

The trace flags can be modified by requests or procedures as well as screen field actions in order to enable or disable all or individual tracing options. Unlike other flags, trace flags retain their value between interactions; they are not re-initialized until application termination which can span multiple interactions. For IMS TM, application termination occurs after all input messages have been processed for a given transaction. For CICS TS, application termination occurs when the pseudo conversation has been terminated which can span multiple transactions.

Note that if execution tracing is not included during APPGEN or the Environment window specifies PRODUCTION (or the ?T statement contains an entry of P) in Trace Control, then tracing is not included in an application and any modifications to the trace flags are ignored. The trace flags cannot be used to enable or disable tracing unless execution tracing has been included during APPGEN with a Trace Control entry of DISABLE ALL or ENABLE ANY (or 0 or 1 on the ?T statement).

Modifications to the trace flags do not affect the tracing options being used until the request or procedure terminates. The new tracing options will become effective upon exit from the request or procedure and remain effective until they are changed or the application terminates. In the case of field actions modifying the flags, the trace options do not become effective until in-storage update processing has been completed, thus becoming effective for the first type M request or procedure. The reason the new trace options do not become effective until execution of a request or procedure completes is to avoid incomplete trace messages being output. For example, if the trace flags were modified after a data base loop had been established within the request or procedure, incomplete and misleading looping and I/O call trace messages could be output.

If it is desired to change the tracing options in effect during the execution of a given request or procedure, then a subroutine request or procedure could be used to modify the appropriate flags. The new tracing options would become effective upon exit from the subroutine and return to the calling request or procedure.

When application execution begins, the trace options in effect are those specified on the Environment window (or ?T statement). Type 1 processing is the earliest point within the interaction cycle for which the trace options in effect can be modified. Therefore, the only way to modify the trace options in effect prior to type 1 processing (e.g., trace messages AF01-AF14), is to modify the trace flags in a previous interaction.

Index

A

Arithmetic expressions, 1-6

ASL

- description, 1-2, 1-11, 1-13, 1-14, 1-15, 1-16, 1-17
- implicit loops, 1-11
- invoking, 1-17
- record processing, 1-16
- segment processing, 1-16
- set operation, 1-11

B

Built-in functions, 2-1, 2-2

C

CALL command (online), 3-2, 3-3

CASE command, 3-4

Character constants, 1-3

Comments, 1-5

Constants, 1-4

CONTINUE command, 3-5

D

Decimal constants, 1-4

DO command, 3-5, 3-7, 3-8

E

ELSE command, 3-9, 3-10

END command, 3-10

F

FIELD command, 3-11

Field length, B-5

Field names, 1-5, B-2

Field type, B-5

Flags

- ATTNID, C-5
- CSTATUS, C-4
- ERROR, C-5
- GOSC, C-6
- HIGHVALU, C-8
- LOWVALU, C-8
- MSGLINE, C-7
- OPERCL, C-7
- OPERID, C-7
- PAGENO, C-9
- partial field, C-8
- RETURNCD, C-4
- SCRNAMEI, C-9
- SCRNAMEO, C-9
- TAPPFLOW, C-2
- TCONTROL, C-2, C-10
- TDLOOP, C-2
- TDSDBIO, C-2

TERMID, C-7
TPROC, C-2
TRANCODE, C-6
TSEGACT, C-2
TSSDBIO, C-2
USERID, C-7

Floating point constants, 1-4

G

GO command, 3-12

I

IF command, 3-13
Implicit loops, 1-11
Integer constants, 1-4
Invoking ASL, 1-17

L

Labels, 1-5
LEAVE command, 3-14, 3-15
LET command, 3-15
Logical expressions, 1-7, 1-8

M

MSG command, 3-16

N

Names, 1-5

O

Output edit, B-5

P

Patterns, 1-4, B-4
PF function, 2-1, 2-2

Q

Qualifiers, B-4

R

Record processing, 1-16
Reserved words, B-2
RETURN command, 3-17

S

Segment processing, 1-16
SET command, 3-18
Set operation, 1-11
Statement layout, 1-9
Statement syntax, 1-9
Syntax of functions and commands, 1-9
System defined names, B-2

T

TRACE command, 3-19
TRANSFER command, 3-20

V

VISION:Inform, 1-6