

Further Programming Aspects

This section covers the following topics:

- End of Program - The END Statement
 - End of Application - The STOP Statement
 - Conditional Processing - The IF Statement
 - Loop Processing
 - Control Breaks
 - Data Computation
 - System Variables and System Functions
 - Stack
 - Processing of Date Information
-

End of Program - The END Statement

The END statement is used to mark the end of a Natural program, subprogram, external subroutine or help routine.

Every one of these objects must contain an END statement as the last statement.

Every object may contain only one END statement.

End of Application - The STOP Statement

The STOP statement is used to terminate the execution of a Natural application. A STOP statement executed anywhere within an application immediately stops the execution of the entire application.

Conditional Processing - The IF Statement

With the IF statement, you define a logical condition, and the execution of the statement attached to the IF statement then depends on that condition.

The IF statement contains three components: IF, THEN, and ELSE.

- In the IF clause, you specify the logical condition which is to be met.
- In the THEN clause you specify the statement(s) to be executed if this condition is met.
- In the (optional) ELSE clause, you can specify the statement(s) to be executed if this condition is *not* met.

So, an IF statement takes the following general form:

```
IF condition
  THEN execute statement(s)
  ELSE execute other statement(s)
END-IF
```

If you wish a certain processing to be performed only if the IF condition is *not* met, you can specify the clause THEN IGNORE, which means that the IF condition will be ignored if it is met.

For more information on logical conditions, see General Information of the Natural Reference documentation.

Example of IF Statement:

```

** Example Program 'IFX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 CITY
  2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
  IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
END-READ
END

```

The IF statement block in the above program causes the following conditional processing to be performed:

- IF the salary is less than 40000, THEN the WRITE statement is to be executed;
- otherwise (ELSE), that is, if the salary is 40000 or more, the DISPLAY statement is to be executed.

The program produces the following output:

NAME	DATE OF BIRTH	ANNUAL SALARY
***** KEEN		SALARY LT 40000
***** FORRESTER		SALARY LT 40000
***** JONES		SALARY LT 40000
***** MELKANOFF		SALARY LT 40000
DAVENPORT	1948-12-25	42000
GEORGES	1949-10-26	182800
***** FULLERTON		SALARY LT 40000

Nested IF Statements

It is possible to use various nested IF statements; for example, you can make the execution of a THEN clause dependent on another IF statement which you specify in the THEN clause.

Example of Nested IF Statements:

```

** Example Program 'IFX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1:1)
  2 BIRTH
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EN=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
      SORTED BY NAME
      IF SALARY (1) LESS THAN 20000
        THEN WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
      ELSE
        IF BIRTH GT #BIRTH
          FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
          DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
          SALARY (1) MAKE (AL=8 IS=OFF)
        END-FIND
      END-IF
    END-IF
  END-IF
SKIP 1
END-FIND
END

```

The above program with nested IF statements produces the following output:

NAME	DATE OF BIRTH	ANNUAL SALARY	MAKE
***** COHEN			SALARY LT 20000
CREMER	1972-12-14	20000	FORD
***** FLEMING			SALARY LT 20000
***** GREENACRE			SALARY LT 20000
PERREAULT	1950-05-12	30500	CHRYSLER
***** SHAW			SALARY LT 20000
STANWOOD	1946-09-08	31000	CHRYSLER FORD

Further Example of IF Statement:

See program IFX03 in library SYSEXP.

Loop Processing

A processing loop is a group of statements which are executed repeatedly until a stated condition has been satisfied, or as long as a certain condition prevails.

Processing loops can be subdivided into database loops and non-database loops:

- *Database processing loops* are those created automatically by Natural to process data selected from a database as a result of a READ, FIND or HISTOGRAM statement. These statements are described in the section Database Access.
- *Non-database processing loops* are initiated by the statements REPEAT, FOR, CALL FILE, CALL LOOP, SORT, and READ WORK FILE.

More than one processing loop may be active at the same time. Loops may be embedded or nested within other loops which remain active (open).

A processing loop must be explicitly closed with a corresponding END-... statement (for example, END-REPEAT, END-FOR, etc.)

The SORT statement, which invokes the sort program of the operating system, closes all active processing loops and initiates a new processing loop.

The following topics are covered below:

- Limiting Database Loops
- Limiting Non-Database Loops - The REPEAT Statement
- Terminating a Processing Loop - The ESCAPE Statement
- Loops Within Loops
- Referencing Statements within a Program

Limiting Database Loops

With the statements READ, FIND, or HISTOGRAM, you have three ways of limiting the number of repetitions of the processing loops initiated with these statements:

- with the session parameter LT,
- with a LIMIT statement,
- or with a limit notation in a READ/FIND/HISTOGRAM statement itself.

LT Session Parameter

With the system command GLOBALS, you can specify the session parameter LT, which limits the number of records which may be read in a database processing loop.

Example:

```
GLOBALS LT=100
```

This limit applies to all READ, FIND and HISTOGRAM statements in the entire session.

LIMIT Statement

In a program, you can use the LIMIT statement to limit the number of records which may be read in a database processing loop.

Example:

```
LIMIT 100
```

The LIMIT statement applies to the remainder of the program unless it is overridden by another LIMIT statement or limit notation.

Limit Notation

With a READ, FIND or HISTOGRAM statement itself, you can specify the number of records to be read in parentheses immediately after the statement name.

Example:

```
READ (10) VIEWXYZ BY NAME
```

This limit notation overrides any other limit in effect, but applies only for the statement in which it is specified.

If the limit set with the LT parameter is smaller than a limit specified with a LIMIT statement or a limit notation, the LT limit has priority over any of these other limits.

Limiting Non-Database Loops - The REPEAT Statement

Non-database processing loops begin and end based on logical condition criteria or some other specified limiting condition.

The REPEAT statement is discussed here as representative of a non-database loop statement.

With the REPEAT statement, you specify one or more statements which are to be executed repeatedly. Moreover, you can specify a logical condition, so that the statements are only executed either until or as long as that condition is met. For this purpose you use an UNTIL or WHILE clause:

- If you specify the logical condition in an UNTIL clause, the REPEAT loop will continue *until* the logical condition is met.
- If you specify the logical condition in a WHILE clause, the REPEAT loop will continue *as long as* the logical condition remains true.

If you specify no logical condition, the REPEAT loop must be exited with an ESCAPE, STOP or TERMINATE statement:

- An ESCAPE statement (see next section) terminates the execution of the processing loop and continues processing outside the loop.
- A STOP statement stops the execution of the entire Natural application.
- A TERMINATE statement stops the execution of the Natural application and also ends the Natural session.

Example of REPEAT Statement:

```

** Example Program 'REPEAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
1 #PAY1 (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  SKIP 1
END-READ
END
    
```

The above program produces the following output:

Page	1	97-08-19 18:42:53	
	NAME	ANNUAL SALARY	#PAY1

	ADKINSON	34500	37950 41745
		33500	36850 40535
		36000	39600 43560
	AFANASSIEV	37000	40700
	ALEXANDER	34500	37950 41745

Terminating a Processing Loop - The ESCAPE Statement

The ESCAPE statement is used to terminate the execution of a processing loop based on a logical condition.

You can place an ESCAPE statement within loops in conditional IF statement groups, in break processing statement groups (AT END OF DATA, AT END OF PAGE, AT BREAK), or as a stand-alone statement implementing the basic logical conditions of a non-database loop.

The ESCAPE statement offers the options TOP and bottom, which determine where processing is to continue after the processing loop has been left via the ESCAPE statement:

- ESCAPE TOP is used to continue processing at the top of the processing loop.
- ESCAPE bottom is used to continue processing with the first statement following the processing loop.

You can specify several ESCAPE statements within the same processing loop.

For further details and examples of the ESCAPE statement, see the Natural Statements documentation.

Loops Within Loops

A database statement can be placed within a database processing loop initiated by another database statement. When database loop-initiating statements are embedded in this way, a "hierarchy" of loops is created, each of which is processed for each record which meets the selection criteria.

Multiple levels of loops can be embedded. For example, non-database loops can be nested one inside the other. Database loops can be nested inside non-database loops. Database and non-database loops can be nested within conditional statement groups.

Example of Nested FIND Statements:

The following program illustrates a hierarchy of two loops, with one FIND loop nested or embedded within another FIND loop.

```

** Example Program 'FINDX06'
  DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 CITY
    2 NAME
    2 PERSONNEL-ID
  1 VEH-VIEW VIEW OF VEHICLES
    2 MAKE
    2 PERSONNEL-ID
  END-DEFINE
  *
  FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
    FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
    DISPLAY NOTITLE NAME CITY MAKE
  END-FIND
  END-FIND
  END

```

The above program selects data from multiple files. The outer FIND loop selects from the EMPLOYEES file all persons who live in New York or Beverley Hills. For each record selected in the outer loop, the inner FIND loop is entered, selecting the car data of those persons from the VEHICLES file. The program produces the following output:

NAME	CITY	MAKE
RUBIN	NEW YORK	FORD
OLLE	BEVERLEY HILLS	GENERAL MOTORS
ADKINSON	BEVERLEY HILLS	FORD
WALLACE	NEW YORK	MAZDA
SPEISER	BEVERLEY HILLS	FORD

Referencing Statements within a Program

Statement reference notation is used to refer to previous statements in a program in order to specify processing over a particular range of data, to override Natural's default referencing (as described for each statement in the Natural Statements documentation, where applicable), or for documentation purposes.

Any Natural statement which causes a processing loop to be initiated and/or causes data elements in a database to be accessed (for example, READ, FIND, HISTOGRAM, SORT, REPEAT, FOR) can be referenced.

When multiple processing loops are used in a program, reference notation is used to uniquely identify the particular database field to be processed by referring back to the statement that originally accessed that field in the database. (If a field can be referenced in such a way, this is indicated in the "Reference Permitted" column of the "Operand Definition Table" in the statement description in the Natural Statements documentation.)

In addition, reference notation can be specified in some statements; for example, AT START OF DATA, AT END OF DATA, AT BREAK and ESCAPE bottom. Without reference notation, an AT START OF DATA, AT END OF DATA or AT BREAK statement will be related to the *outermost* active READ, FIND, HISTOGRAM, SORT or READ WORK FILE loop. With reference notation, you can relate it to another active processing loop.

If reference notation is specified with an ESCAPE bottom statement, processing will continue with the first statement following the processing loop identified by the reference notation.

Statement reference notation may be specified in the form of a *statement label* or a *source-code line number*.

A statement label consists of several characters, the last of which must be a period (.). The period serves to identify the entry as a label.

A statement that is to be referenced is marked with a label by placing the label at the beginning of the line that contains the statement. For example:

```
0030 ...
    0040 READ1. READ VIEWXYZ BY NAME
0050 ...
```

In the statement that references the marked statement, the label is placed in parentheses at the location indicated in the statement's syntax diagram (as described in the Natural Statements documentation). For example:

```
AT BREAK (READ1.) OF NAME
```

If source-code line numbers are used for referencing, they must be specified as 4-digit numbers (leading zeros must not be omitted) and in parentheses. For example:

```
AT BREAK (0040) OF NAME
```

In a statement where the label/line number relates a particular field to a previous statement, the label/line number is placed in parentheses after the field name. For example:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Line numbers and labels can be used interchangeably.

Example with Line Numbers:

The following program uses line numbers for referencing. In this particular example, the line numbers refer to the statements that would be referenced in any case by default.

```
0010 ** Example Program 'LABELX01'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040 2 NAME
0050 2 FIRST-NAME
0060 2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080 2 PERSONNEL-ID
0090 2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140 FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0130)
0150 IF NO RECORDS FOUND
0160 MOVE '***NO CAR***' TO MAKE
0170 END-NOREC
0180 DISPLAY NOTITLE NAME (0130) (IS=ON) FIRST-NAME (0130) (IS=ON)
0190 MAKE (0140)
0200 END-FIND /* (0140)
0210 END-READ /* (0130)
0220 END
```

Example with Labels:

The following example illustrates the use of statement reference labels. It is identical to the previous program, except that labels are used for referencing instead of line numbers.

```
0010 ** Example Program 'LABELX02'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040 2 NAME
0050 2 FIRST-NAME
0060 2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080 2 PERSONNEL-ID
0090 2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140 FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
0150 IF NO RECORDS FOUND
0160 MOVE '***NO CAR***' TO MAKE
0170 END-NOREC
0180 DISPLAY NOTITLE NAME (RD.) (IS=ON) FIRST-NAME (RD.) (IS=ON)
0190 MAKE (FD.)
0200 END-FIND /* (FD.)
0210 END-READ /* (RD.)
0220 END
```

Both programs produce the following output:

NAME	FIRST-NAME	MAKE
JONES	VIRGINIA MARSHA	***NO CAR*** CHRYSLER CHRYSLER
	ROBERT LILLY EDWARD MARTHA LAUREL KEVIN GREGORY	GENERAL MOTORS ***NO CAR*** GENERAL MOTORS ***NO CAR*** GENERAL MOTORS DATSUN FORD
JOPER	MANFRED	***NO CAR***
JOUSSELIN	DANIEL	RENAULT
JUBE	GABRIEL	***NO CAR***
JUNG	ERNST	***NO CAR***
JUNKIN	JEREMY	***NO CAR***
KAISER	REINER	***NO CAR***

Control Breaks

A control break occurs when the value of a control field changes.

The execution of statements can be made dependent on a control break. A control break can also be used for the evaluation of Natural system functions. System functions are discussed later in this section.

- AT BREAK Statement
- Automatic Break Processing
- BEFORE BREAK PROCESSING Statement
- User-Initiated Break Processing - The PERFORM BREAK PROCESSING Statement

AT BREAK Statement

With the statement AT BREAK, you specify the processing which is to be performed whenever a control break occurs, that is, whenever the value of a control field which you specify with the AT BREAK statement changes. As a control field, you can use a database field or a user-defined variable.

Control Break Based on a Database Field

The field specified as control field in an AT BREAK statement is usually a database field.

Example:

```
...
  AT BREAK OF DEPT
    statements
  END-BREAK
...
```

In this example, the control field is the database field DEPT; if the value of the field changes, for example, FROM "SALE01" to "SALE02", the *statements* specified in the AT BREAK statement would be executed.

Instead of an entire field, you can also use only part of a field as a control field. With the notation */n/* you can determine that only the first *n* positions of a field are to be checked for a change in value.

Example:

```
...
  AT BREAK OF DEPT /4/
    statements
  END-BREAK
...
```

In this example, the specified *statements* would only be executed if the value of the first 4 positions of the field DEPT changes, for example, FROM "SALE" to "TECH"; if, however, the field value changes from "SALE01" to "SALE02", this would be ignored and no AT BREAK processing performed.

Example of AT BREAK Statement using a Database Field:

```
** Example Program 'ATBEX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 CITY
    2 COUNTRY
    2 JOB-TITLE
    2 SALARY (1:1)
  END-DEFINE
```

```

*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF CITY
    WRITE / OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X)
      5X 'AVERAGE:' T*SALARY AVER(SALARY(1)) //
      COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END

```

In the above program, the first WRITE statement is executed whenever the value of the field CITY changes. In the AT BREAK statement, the system functions OLD, AVER and COUNT are evaluated (and output in the WRITE statement). In the AT END OF DATA statement, the system function TOTAL is evaluated. The program produces the following output:

Page	1	97-08-19	18:17:27
CITY	NAME	POSITION	SALARY
-----	-----	-----	-----
AIKEN	SENKO	PROGRAMMER	31500
A I K E N	AVERAGE :		31500
	1 RECORDS FOUND		
ALBUQUERQ	HAMMOND	SECRETARY	22000
ALBUQUERQ	ROLLING	MANAGER	34000
ALBUQUERQ	FREEMAN	MANAGER	34000
ALBUQUERQ	LINCOLN	ANALYST	41000
A L B U Q U E R Q U E	AVERAGE :		32750
	4 RECORDS FOUND		
TOTAL (ALL RECORDS):			162500

Control Break Based on a User-Defined Variable

A user-defined variable can also be used as control field in an AT BREAK statement.

In the following program, the user-defined variable #LOCATION is used as control field.

```

** Example Program 'ATBREX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END
    
```

The above program produces the following output:

Page	1		97-08-19 18:21:23
#LOCATION	POSITION	SALARY	
-----	-----	-----	
AIKEN USA	PROGRAMMER	31500	
ALBUQUERQUE USA	SECRETARY	22000	
ALBUQUERQUE USA	MANAGER	34000	
ALBUQUERQUE USA	MANAGER	34000	
ALBUQUERQUE USA	ANALYST	41000	

Multiple Control Break Levels

As explained above, the notation "/n/" allows some portion of a field to be checked for a control break. It is possible to combine several AT BREAK statements, using an entire field as control field for one break and part of the same field as control field for another break. In such a case, the break at the lower level (entire field) must be specified before the break at the higher level (part of field); that is, in the first AT BREAK statement the entire field must be specified as control field, and in the second one part of the field.

The following example program illustrates this, using the field DEPT as well as the first 4 positions of that field (DEPT /4/).

```

** Example Program 'ATBREX03'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 DEPT
    2 SALARY (1:1)
    2 CURR-CODE (1:1)
  END-DEFINE
  READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
                                WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'
    AT BREAK OF DEPT
      WRITE '*** LOWEST BREAK LEVEL ***' /
    END-BREAK
    AT BREAK OF DEPT /4/
      WRITE '*** HIGHEST BREAK LEVEL ***'
    END-BREAK
    DISPLAY DEPT NAME 'POSITION' JOB-TITLE
  END-READ
  END

```

DEPARTMENT CODE	NAME	POSITION
TECH05	HERZOG	MANAGER
TECH05	LAWLER	MANAGER
TECH05	MEYER	MANAGER
*** LOWEST BREAK LEVEL ***		
TECH10	DEKKER	DBA
*** LOWEST BREAK LEVEL ***		
*** HIGHEST BREAK LEVEL ***		

In the following program, one blank line is output whenever the value of the field DEPT changes; and whenever the value in the first 4 positions of DEPT changes, a record count is carried out by evaluating the system function COUNT.

```

** Example Program 'ATBREX04'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 DEPT
    2 REDEFINE DEPT
      3 #GENDEP (A4)
    2 NAME
    2 SALARY (1)

```

```

END-DEFINE
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  AT BREAK OF DEPT
  SKIP 1
END-BREAK
AT BREAK OF DEPT /4/
  WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
END-BREAK
END-READ
END

```

** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **		
DEPT	NAME	SALARY

ADMA01	JENSEN	180000
ADMA01	PETERSEN	105000
ADMA01	MORTENSEN	320000
ADMA01	MADSEN	149000
ADMA01	BUHL	642000
ADMA02	HERMANSEN	391500
ADMA02	PLOUG	162900
ADMA02	HANSEN	234000
8 RECORDS FOUND IN: ADMA		
COMP01	HEURTEBISE	168800
1 RECORDS FOUND IN: COMP		

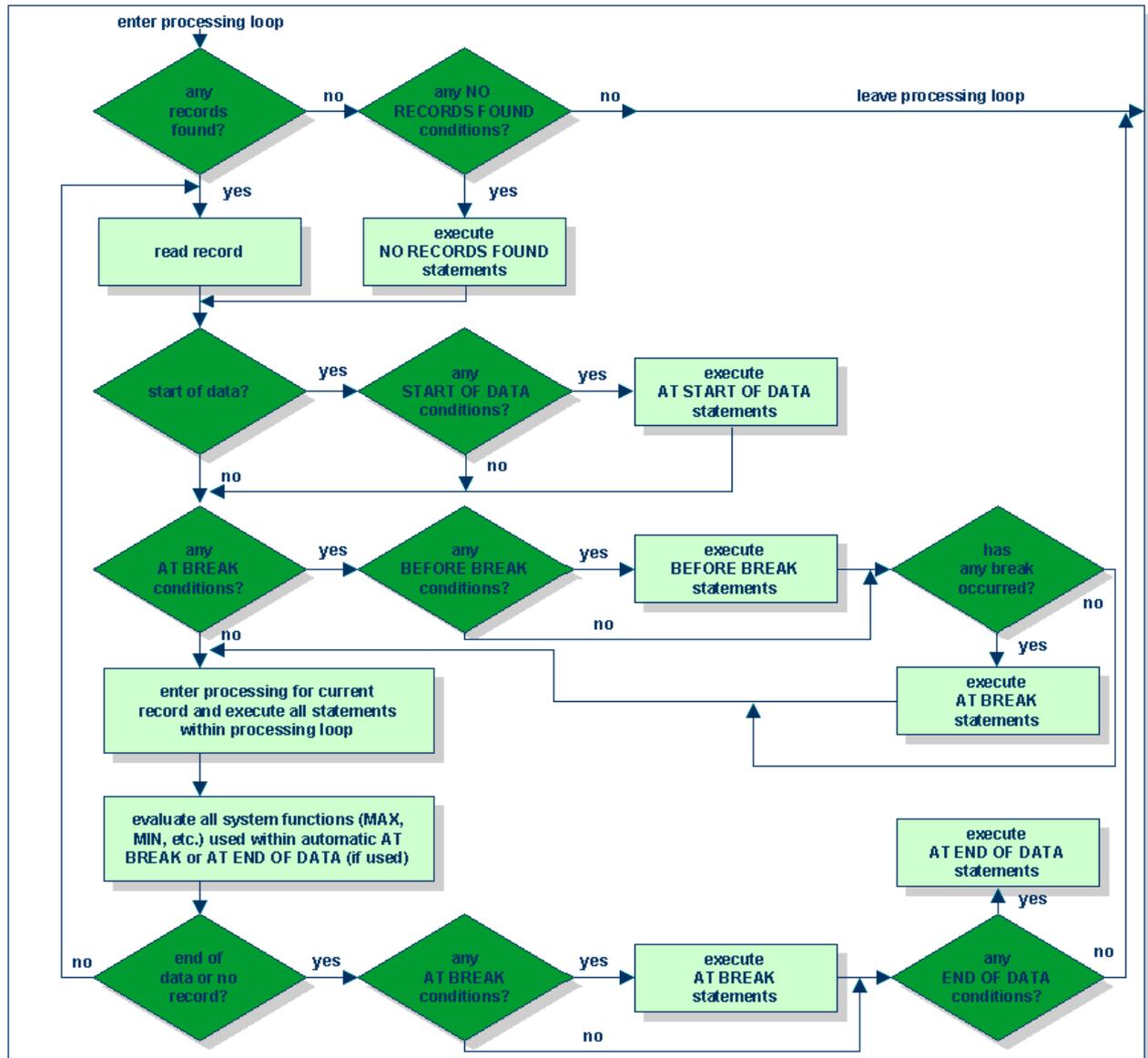
Automatic Break Processing

Automatic break processing is in effect for a FIND, READ, HISTOGRAM, SORT or READ WORK FILE processing loop which contains an AT BREAK statement.

The value of the control field specified with the AT BREAK statement is checked only for records which satisfy the selection criteria of both the WITH clause and the WHERE clause.

Natural system functions (AVER, MAX, MIN, etc.) are evaluated for each record after all statements within the processing loop have been executed. System functions are not evaluated for any record which is rejected by WHERE criteria.

The figure below illustrates the flow logic of automatic break processing.



BEFORE BREAK PROCESSING Statement

With the BEFORE BREAK PROCESSING statement, you can specify statements that are to be executed immediately before a control break; that is, before the value of the control field is checked, before the statements specified in the AT BREAK block are executed, and before any Natural system functions are evaluated.

Example of BEFORE BREAK PROCESSING Statement:

```

** Example Program 'BEFORX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
1 #INCOME (P11)
END-DEFINE
*
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
  BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
    'ANNUAL/INCOME' #INCOME
    'SALARY' SALARY(1) (LC==) / '+ BONUS' BONUS(1,1) (IC==)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-' (24)
  END-BREAK
END-READ
END

```

NAME	FIRST-NAME	ANNUAL INCOME	SALARY + BONUS
BACHMANN	HANS	297546 =	293546 +4000
BAECKER	JOHANNES	420244 =	413644 +6600
BAECKER	KARL	52650 =	48600 +4050
BAGAZJA	MARJAN	152700 =	129700 +23000
BAILLET	PATRICK	198500 =	188000 +10500

User-Initiated Break Processing - The PERFORM BREAK PROCESSING Statement

With automatic break processing, the statements specified in an AT BREAK block are executed whenever the value of the specified control field changes - regardless of the position of the AT BREAK statement in the processing loop.

With a PERFORM BREAK PROCESSING statement, you can perform break processing at a specified position in a processing loop: the PERFORM BREAK PROCESSING statement is executed when it is encountered in the processing flow of the program.

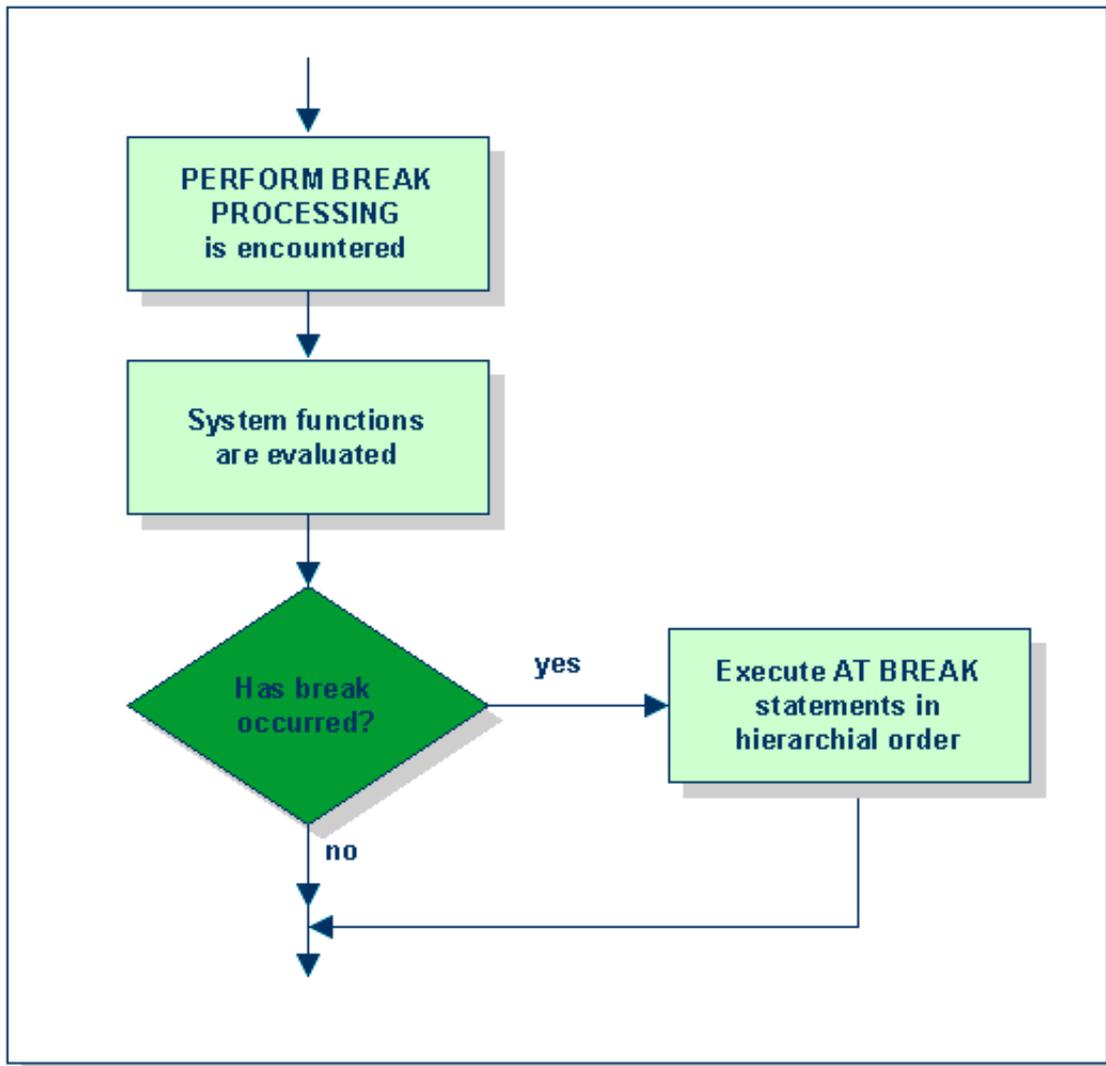
Immediately after the PERFORM BREAK PROCESSING, you specify one or more AT BREAK statement blocks:

```
...  
  PERFORM BREAK PROCESSING  
    AT BREAK OF field1  
      statements  
    END-BREAK  
    AT BREAK OF field2  
      statements  
    END-BREAK  
  ...
```

When a PERFORM BREAK PROCESSING is executed, Natural checks if a break has occurred; that is, if the value of the specified control field has changed; and if it has, the specified statements are executed.

With PERFORM BREAK PROCESSING, system functions are evaluated *before* Natural checks if a break has occurred.

The following figure illustrates the flow logic of user-initiated break processing:



Example of PERFORM BREAK PROCESSING Statement:

```

** Example Program 'PERFBX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
1 #CNTL      (N2)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY DEPT
  AT BREAK OF DEPT          /* <- automatic break processing
  SKIP 1
  WRITE 'SUMMARY FOR ALL SALARIES          '
    'SUM:'  SUM(SALARY(1))
    'TOTAL:' TOTAL(SALARY(1))
  ADD 1 TO #CNTL
END-BREAK
IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* <- user-initiated break processing
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 100000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* <- user-initiated break processing
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 150000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END
    
```

Page	1	97-08-18	17:11:11
	NAME	DEPARTMENT CODE	ANNUAL SALARY

	JENSEN	ADMA01	180000
	PETERSEN	ADMA01	105000
	MORTENSEN	ADMA01	320000
	MADSEN	ADMA01	149000
	BUHL	ADMA01	642000
	SUMMARY FOR ALL SALARIES	SUM:	1396000 TOTAL: 1396000
	SUMMARY FOR SALARY GREATER 100000	SUM:	1396000 TOTAL: 1396000
	SUMMARY FOR SALARY GREATER 150000	SUM:	1142000 TOTAL: 1142000
	HERMANSEN	ADMA02	391500
	PLOUG	ADMA02	162900
	SUMMARY FOR ALL SALARIES	SUM:	554400 TOTAL: 1950400
	SUMMARY FOR SALARY GREATER 100000	SUM:	554400 TOTAL: 1950400
	SUMMARY FOR SALARY GREATER 150000	SUM:	554400 TOTAL: 1696400

Further Example of AT BREAK Statement:

See program ATBREX06 in library SYSEXP.

Data Computation

This section discusses the arithmetic statements COMPUTE, ADD, SUBTRACT, MULTIPLY and DIVIDE; as well as the statements MOVE and COMPRESS, which are used to transfer values from one field to another.

- Format of Fields
- COMPUTE Statement
- Statements MOVE and COMPUTE
- Statements ADD, SUBTRACT, MULTIPLY and DIVIDE
- COMPRESS Statement
- Mathematical Functions

Format of Fields

For optimum processing, user-defined variables used in arithmetic statements should be defined with format P (packed numeric).

COMPUTE Statement

The COMPUTE statement is used to perform arithmetic operations. The following connecting operators are available:

Exponentiation	**
Multiplication	*
Division	/
Addition	+
Subtraction	-

Parentheses may be used to indicate logical grouping.

Example 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

In this example, the value of the field LEAVE-DUE is multiplied by 1.1, and the result is placed in the field LEAVE-DUE.

Example 2:

```
COMPUTE #A = SQRT (#B)
```

In this example, the square root of the value of the field #B is evaluated, and the result is assigned to the field #A. "SQRT" is a mathematical function supported in the arithmetic statements COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE. An overview of mathematical functions is provided later in this section.

Example 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

In this example, the first bonus of the current year and the current salary amount are added and assigned to the field #INCOME.

Statements MOVE and COMPUTE

The statements MOVE and COMPUTE can be used to transfer the value of an operand into one or more fields. The operand may be a constant such as a text item or a number, a database field, a user-defined variable, a system variable, or, in certain cases, a system function.

The difference between the two statements is that in the MOVE statement the value to be moved is specified on the left; in the COMPUTE statement the value to be assigned is specified on the right, as shown in the following examples.

Examples:

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY and DIVIDE

The ADD, SUBTRACT, MULTIPLY and DIVIDE statements are used to perform arithmetic operations.

Examples:

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

All four statements have a ROUNDED option, which you can use if you wish the result of the operation to be rounded.

The Natural Statements documentation provides more detailed information on these statements.

Example of MOVE, SUBTRACT and COMPUTE Statements:

The following program demonstrates the use of user-defined variables in arithmetic statements. It calculates the ages and wages of three employees and outputs these.

```
** Example Program 'COMPUX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY          (1:1)
  2 BONUS           (1:1,1:1)
1 #DATE            (N8)
1 REDEFINE #DATE
  2 #YEAR           (N4)
  2 #MONTH          (N2)
  2 #DAY            (N2)
1 #BIRTH-YEAR     (A4)
1 REDEFINE #BIRTH-YEAR
  2 #BIRTH-YEAR-N (N4)
1 #AGE            (N3)
1 #INCOME         (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
```

```

SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END

```

Page	1		99-01-22	12:42:50
NAME	POSITION	#AGE	#INCOME	
JONES	MANAGER	58	55000	
JONES	DIRECTOR	53	50000	
JONES	PROGRAMMER	43	31000	

COMPRESS Statement

The COMPRESS statement is used to transfer (combine) the contents of two or more operands into a single alphanumeric field.

Leading zeros in a numeric field and trailing blanks in an alphanumeric field are suppressed before the field value is moved to the receiving field.

By default, the transferred values are separated from one another by a single blank in the receiving field. Other separating possibilities are described in the Natural Statements documentation.

Example:

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

In this example, a text constant ('NAME:'), a database field (FIRST-NAME) and a user-defined variable (#LAST-NAME) are combined into one user-defined variable (#FULLNAME) using a COMPRESS statement.

For further information on the COMPRESS statement, please refer to the Natural Statements documentation.

Example of COMPRESS and MOVE Statements:

```

** Example Program 'ComPRX01'
DEFINE DATA LOCAL
1 MYVIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
MOVE NAME TO #LAST-NAME
COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END

```

The above program illustrates the use of the statements MOVE and COMPRESS. Notice the output format of the compressed field:

Page	1		97-08-18	17:47:03
	#FULL-NAME	FIRST-NAME	I	NAME
	=====	-----	-	-----
	NAME: VIRGINIA J JONES	VIRGINIA	J JONES	
	NAME: MARSHA JONES	MARSHA	JONES	
	NAME: ROBERT B JONES	ROBERT	B JONES	

In multiple-line displays, it may be useful to combine fields/text in a user-defined variable by using a COMPRESS statement.

Example of COMPRESS Statement:

In the following program, three user-defined variables are used: #FULLSAL, #FULLNAME, and #FULLCITY. #FULLSAL, for example, contains the text 'SALARY:' and the database fields SALARY and CURR-CODE. The WRITE statement then references only the compressed variables.

```

** Example Program 'COMPRX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
2 NAME
2 FIRST-NAME
2 SALARY (1:1)
2 CURR-CODE (1:1)
2 CITY
2 ADDRESS-LINE (1:1)
2 ZIP
1 #FULLSAL (A25)
1 #FULLNAME (A25)
1 #FULLCITY (A25)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULLSAL
COMPRESS FIRST-NAME NAME INTO #FULLNAME
COMPRESS ZIP CITY INTO #FULLCITY
DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X)
WRITE 1/5 #FULLNAME 1/37 #FULLSAL
2/5 ADDRESS-LINE (1)
3/5 #FULLCITY
SKIP 1
END-READ
END

```

Page	1	97-08-19	18:01:17
NAME AND ADDRESS -----			
R U B I N	SYLVIA RUBIN	SALARY: USD 17000	
	2003 SARAZEN PLACE		
	10036 NEW YORK		
W A L L A C E	MARY WALLACE	SALARY: USD 38000	
	12248 LAUREL GLADE C		
	10036 NEW YORK		
K E L L O G G	HENRIETTA KELLOGG	SALARY: USD 52000	
	1001 JEFF RYAN DR.		
	19711 NEWARK		

Mathematical Functions

The following Natural mathematical functions are supported in arithmetic processing statements (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT).

Function	Meaning
ABS (<i>field</i>)	Absolute value of <i>field</i> .
ATN (<i>field</i>)	Arc tangent of <i>field</i> .
COS (<i>field</i>)	Cosine of <i>field</i> .
EXP (<i>field</i>)	Exponential of <i>field</i> .
FRAC (<i>field</i>)	Fractional part of <i>field</i> .
INT (<i>field</i>)	Integer part of <i>field</i> .
LOG (<i>field</i>)	Natural logarithm of <i>field</i> .
SGN (<i>field</i>)	Sign of <i>field</i> .
SIN (<i>field</i>)	Sine of <i>field</i> .
SQRT (<i>field</i>)	Square root of <i>field</i> .
TAN (<i>field</i>)	Tangent of <i>field</i> .
VAL (<i>field</i>)	Numeric value of an alphanumeric <i>field</i> .

See the Natural Reference documentation for a detailed explanation of each mathematical function.

Further Examples of COMPUTE, MOVE and COMPRESS Statements:

See programs WRITEX11, IFX03 and COMPRX03 in library SYSEXPG.

System Variables and System Functions

The following topics are covered below:

- System Variables
- System Functions

System Variables

Natural system variables contain information about the current Natural session, such as: the current library, the user and terminal identification; the current status of a loop processing; the current report processing status; the current date and time.

This information may be used in Natural programs by specifying the appropriate system variables. For example:

System Variable	Content
*INIT-USER	The user ID of the terminal user.
*LANGUAGE	The language in effect.
*LIBRARY-ID	The current library ID.
*INIT-ID	The terminal ID.
*ERROR-NR	The Natural error number.
*PAGE-NUMBER	The current value for page number.
*COUNTER	The number of times a processing loop has been entered.
*NUMBER	The number of records selected.

Some date and time system variables include the following:

System Variable	Content
*DATU	Current date in format MM/DD/YY
*DAT4U	Current date in format MM/DD/YYYY
*DATE	Current date in format DD/MM/YY
*DAT4E	Current date in format DD/MM/YYYY
*DATI	Current date in format YY-MM-DD
*DAT4I	Current date in format YYYY-MM-DD
*DATD	Current date in format DD.MM.YY
*DAT4D	Current date in format DD.MM.YYYY
*TIME	Time of day in format HH:MM:SS.T
*TIMN	Time of day in format HHMMSSST

The names of all system variables begin with an asterisk (*).

Date and time system variables may be specified in a DISPLAY, WRITE, PRINT, MOVE or COMPUTE statement.

For further information on system variables, see System Variables in the Natural Reference documentation.

System Functions

Natural system functions are a set of statistical and mathematical functions that can be applied to the data after a record has been processed but before break processing occurs.

System functions may be specified in a WRITE, DISPLAY, PRINT, COMPUTE or MOVE statement that is used in conjunction with an AT END OF PAGE, AT END OF DATA or AT BREAK statement.

In the case of an AT END OF PAGE statement, the corresponding DISPLAY statement must include the GIVE SYSTEM FUNCTIONS clause (as shown in the example below).

The following system functions are available:

System Function	Information Returned
AVER (<i>field</i>)	Average of all values for <i>field</i> .
NAVER (<i>field</i>)	Average of all values for <i>field</i> , not counting null values.
MAX (<i>field</i>)	Maximum value of <i>field</i> .
MIN (<i>field</i>)	Minimum value of <i>field</i> .
NMIN (<i>field</i>)	Minimum value of <i>field</i> , not counting null values.
OLD (<i>field</i>)	Value of <i>field</i> value prior to change in control value (AT BREAK condition).
SUM (<i>field</i>)	Sum of all <i>field</i> values (reset when control value in AT BREAK changes).
TOTAL (<i>field</i>)	Total of all <i>field</i> values (not reset when control value in AT BREAK changes).
COUNT (<i>field</i>)	Number of passes through a processing loop.
NCOUNT (<i>field</i>)	Number of passes through a processing loop, not counting passes where the control <i>field</i> contains a null value.

For further information on system functions, see Natural System Functions in the Natural Reference documentation.

Example of System Variables and System Functions:

```

** Example Program 'SYSVAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:1)
  3 CURR-CODE
  3 SALARY
  3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'EMPLOYEE SALARY REPORT AS OF' *DATE /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
    NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER(SALARY(1))
END-ENDPAGE
END

```

The above program illustrates the use of system variables and system functions:

The system variable *DATE is output with the WRITE TITLE statement; the system variable *TIME is output with the AT START OF DATA statement.

The system function OLD is used in the AT END OF DATA statement; the system function AVER is used in the AT END OF PAGE statement.

Note how the system variables and system function are displayed:

EMPLOYEE SALARY REPORT AS OF 18/01/1999				
NAME	CURRENT POSITION	INCOME		
		CURRENCY CODE	ANNUAL SALARY	BONUS

REPORT CREATED AT: 11:51:29.3 HOURS				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY: 31333				

Further Examples of System Variables:

See programs EDITMX05, READX04 and WTITLX01 in library SYSEXP.

Further Examples of System Functions:

See programs ATBREX06 and ATENPX01 in library SYSEXP.

Stack

The Natural stack is a kind of "intermediate storage" in which you can store Natural commands, user-defined commands, and input data to be used by an INPUT statement. In the stack you can store a series of functions which are frequently executed one after the other, such as a series of logon commands.

The data/commands stored in the stack are "stacked" on top of one another. You can decide whether to put them on top or at the bottom of the stack. The data/command in the stack can only be processed in the order in which they are stacked, beginning from the top of the stack.

In a program, you may reference the system variable *DATA to determine the content of the stack (see the Natural Reference documentation for further information).

The total size of the stack is defined by the remaining portion in the ESIZE buffer after allocation for the global data area and the program source area.

The following topics are covered below:

- Stack Processing
- Placing Data in the Stack
- Clearing the Stack

Stack Processing

The processing of the commands/data stored in the stack differs depending on the function being performed.

If a command is expected, that is, the NEXT prompt is about to be displayed, Natural first checks if a command is on the top of the stack. If there is, the NEXT prompt is suppressed and the command is read and deleted from the stack; the command is then executed as if it had been entered manually in response to the NEXT prompt.

If an INPUT statement containing input fields is being executed, Natural first checks if there are any input data on the top of the stack. If there are, these data are passed to the INPUT statement (in delimiter mode); the data read from the stack must be format-compatible with the variables in the INPUT statement; the data are then deleted from the stack.

If an INPUT statement was executed using data from the stack, and this INPUT statement is re-executed via a REINPUT statement, the INPUT statement screen will be re-executed displaying the same data from the stack as when it was executed originally. With the REINPUT statement, no further data are read from the stack.

When a Natural program terminates normally, the stack is flushed beginning from the top until either a command is on the top of the stack or the stack is cleared. When a Natural program is terminated via the terminal command "%%" or with an error, the stack is cleared entirely.

Placing Data in the Stack

The following methods can be used to place data/commands on the stack:

STACK Parameter

The Natural profile parameter `STACK` may be used to place data/commands on the stack. The `STACK` parameter, which is described in the Natural Operations documentation, can be specified by the Natural administrator in the Natural parameter module at the installation of Natural; or you can specify it as a dynamic parameter when you invoke Natural.

When data/commands are to be placed on the stack via the `STACK` parameter, multiple commands must be separated from one another by a semicolon (;). If a command is to be passed within a sequence of data or command elements, it must be preceded by a semicolon.

Data for multiple `INPUT` statements must be separated from one another by a colon (:). Data that are to be read by a separate `INPUT` statement must be preceded by a colon. If a command is to be stacked which requires parameters, no colon is to be placed between the command and the parameters.

Semicolon and colon must not be used within the input data themselves as they will be interpreted as separation characters.

STACK Statement

The `STACK` statement can be used within a program to place data/commands in the stack. The data elements specified in one `STACK` statement will be used for one `INPUT` statement, which means that if data for multiple `INPUT` statements are to be placed on the stack, multiple `STACK` statements must be used.

Data may be placed on the stack either unformatted or formatted:

- If unformatted data are read from the stack, the data string is interpreted in delimiter mode and the characters specified with the session parameters `IA` (Input Assignment character) and `ID` (Input Delimiter character) are processed as control characters for keyword assignment and data separation.
- If formatted data are placed on the stack, each content of a field will be separated and passed to one input field in the corresponding `INPUT` statement.

See the Natural Statements documentation for further information on the `STACK` statement.

FETCH and RUN Statements

The execution of a `FETCH` or `RUN` statement that contains parameters to be passed to the invoked program will result in these parameters being placed on top of the stack.

Clearing the Stack

The contents of the stack can be deleted with the `RELEASE` statement. See the Natural Statements documentation for details on the `RELEASE` statement.

Processing of Date Information

This section covers various aspects concerning the handling of dates in your Natural applications:

- Edit Masks for Date Fields and Date System Variables
- Default Edit Mask for Date - The DTFORM Parameter
- Date Format for Alphanumeric Representation - The DF Parameter
- Date Format for Output - The DFOUT Parameter
- Date Format for Stack - The DFSTACK Parameter
- Year Sliding Window - The YSLW Parameter
- Combinations of DFSTACK and YSLW
- Date Format for Default Page Title - The DFTITLE Parameter

Edit Masks for Date Fields and Date System Variables

If you wish the value of a date field to be output in a specific representation, you usually specify an edit mask for the field. With an edit mask, you determine character by character what the output is to look like.

If you wish to use the current date in a specific representation, you need not define a date field and specify an edit mask for it; instead you can simply use a *date system variable*. Natural provides various date system variables, which contain the current date in different representations. Some of these representations contain a 2-digit year component, some a 4-digit year component.

For more information see the examples of date system variables. For more information and a list of all date system variables, see the Natural Reference documentation.

Default Edit Mask for Date - The DTFORM Parameter

The profile parameter DTFORM determines the default format used for dates as part of the default title on Natural reports, for date constants and for date input.

This date format determines the sequence of the day, month and year components of a date, as well as the delimiter characters to be used between these components.

Possible DTFORM settings are:

Setting	Date Format*	Example
DTFORM=I	<i>yyyy-mm-dd</i>	1997-12-31
DTFORM=G	<i>dd.mm.yyyy</i>	31.12.1997
DTFORM=E	<i>dd/mm/yyyy</i>	31/12/1997
DTFORM=U	<i>mm/dd/yyyy</i>	12/31/1997

* *dd* = day, *mm* = month, *yyyy* = year.

The DTFORM parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. By default, DTFORM=I applies.

Date Format for Alphanumeric Representation - The DF Parameter

The session parameter DF only applies to date fields for which no edit mask is specified.

If an edit mask is specified, the representation of the field value is determined by the edit mask. If no edit mask is specified, the representation of the field value is determined by the session parameter DF in combination with the DTFORM profile parameter.

With the DF parameter, you can choose one of the following date representations:

DF=S	8-byte representation with 2-digit year component and delimiters (<i>yy-mm-dd</i>).
DF=I	8-byte representation with 4-digit year component without delimiters (<i>yyyymmdd</i>).
DF=L	10-byte representation with 4-digit year component and delimiters (<i>yyyy-mm-dd</i>).

For each representation, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

By default, DF=S applies (except for INPUT statements; see below).

The DF parameter is evaluated at compilation. It can be specified with the FORMAT statement, the statements INPUT, DISPLAY, WRITE and PRINT (at statement and field level), and the statements MOVE, COMPRESS, STACK, RUN and FETCH (at field level).

The DF parameter applies to the following:

- **DISPLAY, WRITE and PRINT:** When the value of a date variable is output with one of these statements, the value is converted to an alphanumeric representation before it is output. The DF parameter determines which representation is used.
- **MOVE and COMPRESS:** When the value of a date variable is transferred to an alphanumeric field with a MOVE or COMPRESS statement, the value is converted to an alphanumeric representation before it is transferred. The DF parameter determines which representation is used.
- **STACK, FETCH and RUN:** When the value of a date variable is placed on the stack, it is converted to alphanumeric representation before it is placed on the stack. The DF parameter determines which representation is used.

The same applies when a date variable is specified as a parameter in a FETCH or RUN statement (as these parameters are also passed via the stack).

- **INPUT:** When a data variable is used in an INPUT statement, the DF parameter determines how a value must be entered in the field.
However, when a date variable for which *no* DF parameter is specified is used in an INPUT statement, the date can be entered either with a 2-digit year component and delimiters or with a 4-digit year component and no delimiters. In this case, too, the sequence of the day, month and year components, and the delimiter characters to be used, are determined by the DTFORM parameter.

With DF=S, only 2 digits are provided for the year information; this means that if a date value contained the century, this information would be lost during the conversion. To retain the century information, you set DF=I or DF=L.

Examples of DF Parameter with WRITE Statements:

```

/* DF=S (default)
WRITE *DATX /* Output has this format: dd.mm.yy
END

FORMAT DF=I
WRITE *DATX /* Output has this format: ddmmyyyy
END

FORMAT DF=L
WRITE *DATX /* Output has this format: dd.mm.yyyy
END

```

These examples assume that DTFORM=G applies.

Example of DF Parameter with MOVE Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'31/12/1997'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE TO #ALPHA /* Result: #ALPHA contains 31/12/97
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31121997
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/12/1997
...

```

This example assumes that DTFORM=E applies.

Example of DF Parameter with STACK Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1 (A10)
  1 #ALPHA2 (A10)
  1 #ALPHA3 (A10)
END-DEFINE
...
STACK TOP DATA #DATE (DF=S) #DATE (DF=I) #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2 #ALPHA3
...
/* Result: #ALPHA1 contains 97-12-31
/*          #ALPHA2 contains 19971231
/*          #ALPHA3 contains 1997-12-31
...

```

This example assumes that DTFORM=I applies.

Example of DF Parameter with INPUT Statement:

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
      #DATE4      /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

This example assumes that DTFORM=I applies.

Date Format for Output - The DFOUT Parameter

The session/profile parameter DFOUT only applies to date fields in INPUT, DISPLAY, PRINT and WRITE statements for which no edit mask is specified, and for which no DF parameter applies.

For date fields which are displayed by INPUT, DISPLAY, PRINT and WRITE statements and for which neither an edit mask is specified nor a DF parameter applies, the profile/session parameter DFOUT determines the format in which the field values are displayed.

Possible DFOUT settings are:

DFOUT=S	Date variables are displayed with a 2-digit year component, and delimiters as determined by the DTFORM parameter (<i>yy-mm-dd</i>).
DFOUT=I	Date variables are displayed with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

By default, DFOUT=S applies. For either DFOUT setting, the sequence of the day, month and year components in the date values is determined by the DTFORM parameter.

The lengths of the date fields are not affected by the DFOUT setting, as either date value representation fits into an 8-byte field.

The DFOUT parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  END-DEFINE
  ...
  WRITE #DATE          /* Output if DFOUT=S is set ...: 97-12-31
                        /* Output if DFOUT=I is set ...: 19971231
  WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 1997-12-31
  ...

```

This example assumes that DTFORM=I applies.

Date Format for Stack - The DFSTACK Parameter

The session/profile parameter DFSTACK only applies to date fields used in STACK, FETCH and RUN statements for which no DF parameter has been specified.

The DFSTACK parameter determines the format in which the values of date variables are placed on the stack via a STACK, RUN or FETCH statement.

Possible DFSTACK settings are:

DFSTACK=S	Date variables are placed on the stack with a 2-digit year component, and delimiters as determined by the profile DTFORM parameter (<i>yy-mm-dd</i>).
DFSTACK=C	Same as DFSTACK=S. However, a change in the century will be intercepted at runtime.
DFSTACK=I	Date variables are placed on the stack with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

By default, DFSTACK=S applies. DFSTACK=S means that when a date value is placed on the stack, it is placed there without the century information (which is lost). When the value is then read from the stack and placed into another date variable, the century is either assumed to be the current one or determined by the setting of the YSLW parameter (see below). This might lead to the century being different from that of the original date value; however, Natural would not issue any error in this case.

DFSTACK=C works the same as DFSTACK=S in that a date value is placed on the stack without the century information. However, if the value is read from the stack and the resulting century is different from that of the original date value (either because of the YSLW parameter, or the original century not being the current one), Natural issues a runtime error.

Note:

This runtime error is already issued at the time when the value is placed on the stack.

DFSTACK=I allows you to place a date value on the stack in a length of 8 bytes without losing the century information.

The DFSTACK parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1 (A8)
  1 #ALPHA2 (A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Result if DFSTACK=S or =C is set: #ALPHA1 contains 97-12-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 19971231
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 1997-12-31
...

```

This example assumes that DTFORM=I and YSLW=0 apply.

Year Sliding Window - The YSLW Parameter

The profile parameter YSLW allows you determine the century of a 2-digit year value.

The YSLW parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. It is evaluated at runtime when an alphanumeric date value with a 2-digit year component is moved into a date variable. This applies to data values which are:

- used with the mathematical function VAL,
- used with the IS(D) option in a logical condition,
- read from the stack as input data, or
- entered in an input field as input data.

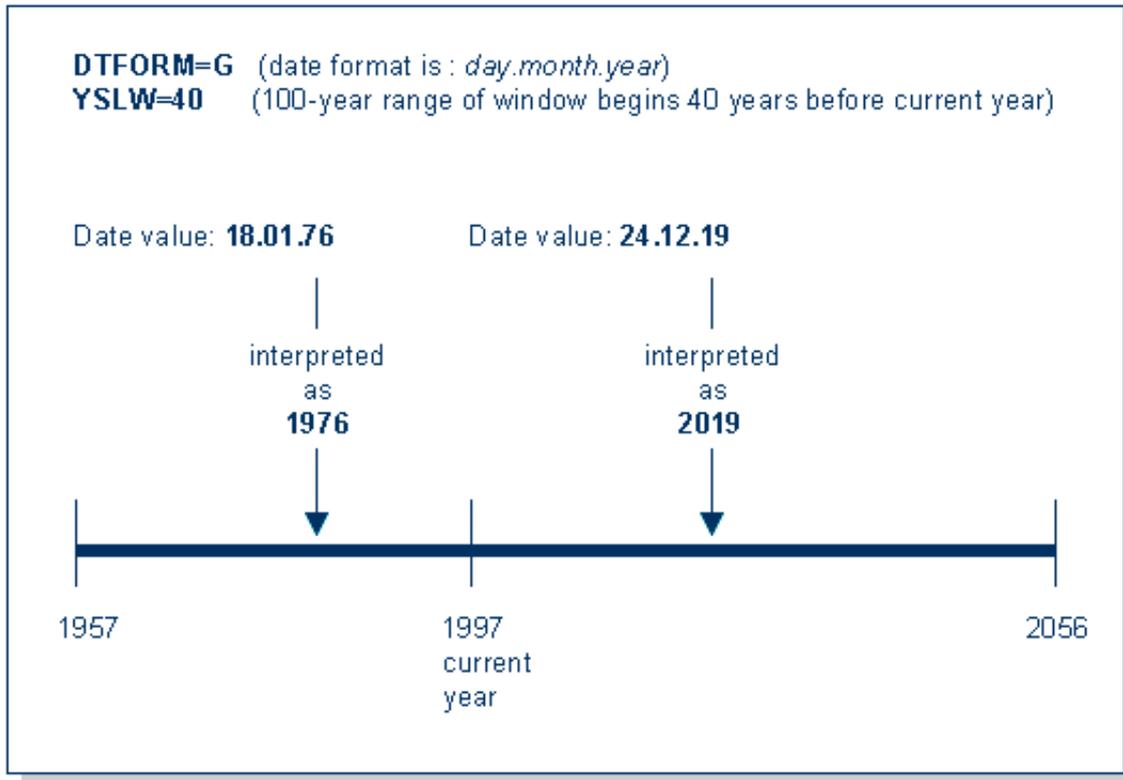
The YSLW parameter determines the range of years covered by a so-called "year sliding window". The sliding-window mechanism assumes a date with a 2-digit year to be within a "window" of 100 years. Within these 100 years, every 2-digit year value can be uniquely related to a specific century.

With the YSLW parameter, you determine how many years in the past that 100-year range is to begin: The YSLW value is subtracted from the current year to determine the first year of the window range.

Possible values of the YSLW parameter are 0 to 99. The default value is YSLW=0, which means that no sliding-window mechanism is used; that is, a date with a 2-digit year is assumed to be in the current century.

Example 1:

If the current year is 1997 and you specify YSLW=40, the sliding window will cover the years 1957 to 2056. A 2-digit year value *nn* from 57 to 99 is interpreted accordingly as 19*nn*, while a 2-digit year value *nn* from 00 to 56 is interpreted as 20*nn*.



Combinations of DFSTACK and YSLW

The following examples illustrate the effects of using various combinations of the parameters DFSTACK and YSLW.

All these examples assume that DTFORM=I applies.

Example 1:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=S (default)

YSLW=20

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 2056
...
/* Result: #DATE2 contains 2056-12-31
```

In this case, the year sliding window is not set appropriately, so that the century information is (inadvertently) changed.

Example 2:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=S (default)

YSLW=50

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
```

In this case, the year sliding window is set appropriately, so that the original century information is correctly restored.

Example 3:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=C

YSLW=0 (default)

```

DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* 56 is assumed to be in current century -> 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)

```

In this case, the century information is (inadvertently) changed. However, this change is intercepted by the DFSTACK=C setting.

Example 4:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=C

YSLW=20

```

DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 2056
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)

```

In this case, the century information is changed due to the year sliding window. However, this change is intercepted by the DFSTACK=C setting.

Date Format for Default Page Title - The DFTITLE Parameter

The session/profile parameter DFTITLE determines the format of the date in a default page title (as output with a DISPLAY, WRITE or PRINT statement).

DFTITLE=S	The date is output with a 2-digit year component and delimiters (<i>yy-mm-dd</i>).
DFTITLE=L	The date is output with a 4-digit year component and delimiters (<i>yyyy-mm-dd</i>).
DFTITLE=I	The date is output with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

For each of these output formats, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

The DFTITLE parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```
WRITE 'HELLO'
  END
/*
/* Date in page title if DFTITLE=S is set ...: 98-10-31
/* Date in page title if DFTITLE=L is set ...: 1998-10-31
/* Date in page title if DFTITLE=I is set ...: 19981031
```

This example assumes that DTFORM=I applies.

Note:

The DFTITLE parameter has no effect on a user-defined page title as specified with a WRITE TITLE statement.