CA Ideal™ for CA Datacom®

Programming Guide

Version 14.02

ca technologies

CA Technologies Product References

This document references the following CA products:

- CA Datacom®/AD
- CA Datacom®/DB
- CA Datacom® CICS Services
- CA Ideal™ for Datacom® (CA Ideal)
- CA Ideal™ for DB2
- CA Ideal™ for VSAM

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: Procedure Definition Language Concepts and Language Elements

The CA Ideal *Programming Reference Guide* describes the Procedure Definition Language (PDL) statements and functions, and symbolic debugger commands. It includes the syntax and a description of each.

CA Ideal *statements* are entered in PDL programs and executed. Statements include FOR, LIST, MOVE, and SELECT.

CA Ideal *functions* are also entered in PDL programs. All functions begin with a dollar sign character ($), for example, $DATE, $EDIT, and $STRING.

Symbolic Debugger commands are entered after initiating a debug session. These commands can be entered interactively from the command line or line command area or in a command member.

PDL Language Elements

The terms defined in this section describe CA Ideal statements, functions, and commands, and the elements of the Procedure Definition Language.

Condition

A condition is one of the following:

- AND/OR condition
- Boolean function
- Condition-Name Flag
- NOT condition
- NULL expression
- Relational-expression
- Search condition
- Where condition

AND/OR Condition

You can combine conditional expressions using AND and OR as following:

```
condition [{AND|OR} condition]…
```

Example

```
IF A=B AND C<D AND E>1
IF FOUND AND NOT RED
IF NOT $NUMERIC (AMOUNT) OR LIMIT>100
```

You can mix NOT, AND, and OR with parentheses to indicate the order in which to apply them. If you do not use parentheses, ANDs are evaluated before ORs. For example, the first expression is equivalent to the next expression,

```
IF (A=B AND C=D) OR E>F
```

```
IF A=B AND C=D OR E>F
```

And the following statement shows that the parentheses override the ¬, default of AND taking precedence over OR.

```
IF A=B AND (C=D OR E>F)
```

You can use the characters (¬), & and (|) interchangeably for NOT, AND, and OR, respectively. Therefore the first expression is equivalent to the next expression,

```
IF A=B AND (C=D OR E>F)
```

```
IF A=B & (C=D | E>F)
```

Combinations of conditional expressions are evaluated according to the following truth tables. (T=True, F=False, and U=Unknown conditions.)

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Thus, T and U yield U; F and U yield F; U and U yield U.

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Thus, T or U yields T; F or U yields U; U or U yields U.

Implied Subjects and Relational Operators

Implied subjects and implied relational operators are also valid. In the following example, A is the implied subject following the OR in the first expression:

```
A = 'B' OR > 'C' is equivalent to A = 'B' OR A > 'C'
```

In the following case, both the subject, A, and the relational-operator, =, are implied following the OR of the first expression:

```
A = 'B' OR 'C' is equivalent to A = 'B' OR A = 'C'
```

You can use the IS NULL expression with an implied subject but cannot use it as an implied relational operator. Thus:

```
A = 'B' OR IS NULL is allowed.
A = 'B' OR IS NULL OR = 'C' is allowed.
```

The following is not allowed:

```
A IS NULL OR B
```

You cannot use an implied subject or implied relational operator in the search condition of a FOR construct against a dataview for SQL access.

Boolean Function

A Boolean function can evaluate to a value of True, False, or Unknown. Boolean functions that can evaluate to Unknown include $ALPHABETIC, $NUMERIC, $VERIFY, and $VERIFY-DATE.

For example, the built-in function to determine if an alphanumeric field has valid numeric content, evaluates the Boolean value to True if field1 has a valid numeric value, a value of False if it does not, and a value of Unknown if the value is null.

```
$NUMERIC(*field1*)
```

Therefore, the following conditional statement is satisfied if the function evaluates to True.

```
IF $NUMERIC(field1)
```

In the statement, the condition is satisfied if the value returned by the function is False.

```
IF NOT $NUMERIC(field2)
```

If *field1* or *field2* in the preceding statements is null, the function evaluates to Unknown and the condition is not satisfied. The effect of Unknown conditions on IF, SELECT, and LOOP statements is described with each statement.

ConditionName Flag

The 1 to 32 character name of the condition that exists when a designated field has a given value. Condition names are defined with a type of C.

You can define RED, YELLOW, and BLUE as condition names subordinate to W_HUE with values of R, Y, and B, respectively. For example, consider the following working data definition:

```
    1     W_HUE       X 1
            RED       C       'R'
            YELLOW    C       'Y'
            BLUE      C       'B'
```

RED is true when W_HUE='R', YELLOW is true when W_HUE='Y', and BLUE is true when W_HUE='B', as shown in the following chart.

Values	**Condition Names**					
W_HUE	RED	BLUE	YELLOW	NOT RED	NOT BLUE	NOT YELLOW
'R'	T	F	F	F	T	T
'Y'	F	F	T	T	T	F
'B'	F	T	F	T	F	T

If W_HUE is defined as a *nullable field*, then you can define NO_HUE as a condition with the value NULL. For example, consider the following working data definition.

```
1       W_HUE        X 1
          RED        C      'R'
          YELLOW     C      'Y'
          BLUE       C      'B'
          NO_HUE     C      NULL
```

NO_HUE is true if W_HUE is NULL. Otherwise, it is false. If RED is defined with the value 'R', RED is true when W_HUE='R', unknown if W_HUE IS NULL, and false otherwise, as shown in the following chart.

Values	**Condition Names**						
W_HUE	RED	BLUE	YELLOW	NOT RED	NOT BLUE	NOT YELLOW	NO_HUE
'R'	T	F	F	F	T	T	F
'Y'	F	F	T	T	T	F	F
'B'	F	T	F	T	F	T	F
NULL	U	U	U	U	U	U	T

Condition names cannot be subordinate to date fields or flags. Like field and group names, they must be qualified if they are not unique. If the field where a condition name is subordinate is subscripted, the condition name also must be subscripted.

Flag

A field with type F can have only the value true or false.

Example

```
IF FOUND                                    IF NOT FOUND
```

In the preceding statement, FOUND was defined as a flag that returns a value of true or false.

NOT Condition

Negates a conditional expression.

Example

```
IF NOT (TYPE = 'A' AND COLOR = 'BLUE')
IF NOT (A = B)
```

The negation of an unknown condition yields an unknown result.

NULL Expression

A conditional expression that tests for null values.

The expression has the following format:

operand `[IS] NULL`

The operand can be any of the following:

- A numeric field, date field, or alphanumeric field defined as nullable in Working Data, Parameter Data, or a dataview definition.
- An arithmetic expression, numeric function, or alphanumeric function with a nullable operand. For example:

  ```
  IF STATE IS NULL
  ```

  The expression is True if the operand evaluates to the null value. Otherwise, the expression is False.

  You can also specify “operand is NOT NULL”. This expression is False if the operand evaluates to the null value. Otherwise, the expression is True.

RelationalExpression

A relational expression is a condition where two operands are compared using a relational operator, yielding a value of true, false, or unknown. A relational expression yields a value of unknown if the value of either operand is null.

Operators

A relational operator can be any of the following:

=	EQ	EQUAL		
¬=	NE	NOT EQUAL	NOT=	
>	GT	GREATER [THAN]		
>=	GE	NOT LESS	NOT<	¬<
<	LT	LESS [THAN]		
<=	LE	NOT GREATER	NOT>	¬>
CONTAINS	NOT CONTAINS			

For all of the relational operators, the symbol ¬ can replace NOT, = or EQ can replace EQUAL, > or GT can replace GREATER, and you can use < or LT LESS. The reserved word THAN after GREATER or LESS is optional. You can add it for clarity. You can add the reserved word IS before any operator for clarity.

Operands

An operand can be a numeric expression (numeric field, date field, arithmetic expression, numeric literal, numeric function), an alphanumeric expression (alphanumeric field, alphanumeric function, alphanumeric literal, or an alpha group), or a non-alpha group.

When two alphanumeric items (including variable length items) are compared, if they differ in length, the shorter item is padded with spaces on the right.

You can compare an alphanumeric expression to a numeric expression; however, a warning is issued. A $NUMBER function is automatically applied to the alphanumeric expression and a numeric comparison is done. If the alphanumeric expression does not contain a valid numeric value, a runtime error occurs.

You can also use non-alpha groups (except restricted non-alpha groups) as operands in simple relations in conditional expressions. The non-alpha group is treated as an alphanumeric field whose length is the size of the group. Subordinate fields are not converted; the hexadecimal contents are compared.

You can only use the relational operators CONTAINS and NOT CONTAINS in FOR constructs for CA Datacom/DB native access dataviews.

You can only use LIKE, IN, BETWEEN, and NOT LIKE, NOT IN, and NOT BETWEEN in where conditions in FOR constructs for SQL dataviews.

Search Condition

A search condition is used in a FOR construct to access a database using SQL. A search condition conforms to the SQL syntax with the exceptions described with the FOR statement for SQL.

Where Condition

A where condition is used in a FOR construct to determine which record or row to access. Qualifications on its use are described with the FOR statement for each respective type of dataview.

Mask Character

A mask character is used in a where condition CONTAINS clause (in a FOR statement for CA Datacom/DB native command access only) to mark the position of a character to ignore in the comparison of a character string. For example, if an asterisk (*) is specified as a mask character, then *AB* represents "any character followed by AB followed by any character". For more information about how to specify mask characters, see the $FIXED-MASK function in the "Symbolic Debugger commands" chapter.

The default mask character is an asterisk (*).

Data Item

A data item is a field, group, or literal.

Data Types

CA Ideal supports the following data types:

- Alphanumeric (type X)
- Variable length alphanumeric (type V)
- Signed numeric (type N)
- Unsigned numeric (type U)
- Date (type D)
- Flag (type F)
- Condition (type C)

Internal Format of Data Items

For alphanumeric (type X) data items, field size is simply the number of characters. For variable length alphanumeric (type V) data items, field size is the number of characters plus two. However, for numeric and date data items, the field size depends on the internal type.

Generally, you do not need to know the internal representation of numeric data items. CA Ideal handles all storage and conversion operations automatically. (See the $STRING and $EDIT functions for details.) However, you might need to know the internal representation of numeric items when using identical parameter matching or to reserve enough space to pass numeric data to a non-ideal subprogram. You might also need to know the internal format when copying a non-alphanumeric group into an alphanumeric field or group by a SET or MOVE statement or a REDEFINITION.

For types N, U, and D, the Digits field indicates the number of digits (base-10) the field can contain. The total number of digits is the sum of the integer places and the decimal (fraction) places.

For example, a digits specification of 7.2 means 7 integer places and 2 decimal (fraction) places, for a total of 9 digits. Thus, for the CA Ideal numeric fields described as follows:

```
=>


---------------------------------------- -----
IDEAL: PARAMETER DEFINITION  PGM DEMO$01 (001) TEST          SYS: DEM  FILL-IN

COMMAND LEVEL FIELD NAME         T I CH/DG OCCUR  U M  COMMENTS/DEP ON/COPY
 ------  ----- ------------------ - - ----- -----
 ======  ===== ====== T O P ====== = = ===== =====  = =  =======================
 ......  1     IDEAL-SIGN-NUM      N   5            U D
 ......  1     IDEAL-UNSIGN-NU     U   5            U D
 ......  1     IDEAL-DATE-NUM      D   5            U D
```

IDEAL-SIGN-NUM can contain any five-digit number from -99,999 to +99,999.

IDEAL-UNSIGN-NUM can contain any five-digit number from 0 to 99,999.

IDEAL-DATE-NUM can contain any five-digit number from -99,999 to 99,999 (representing 273 years from the base year).

The CA Ideal internal formats for numeric and date types follow:

Z	Zoned decimal
P	Packed decimal
B	Binary

For full descriptions, see *IBM System/370 Principles of Operation*.

For all three internal formats, CA Ideal still uses the convention that indicates the number of decimal (base-10) digits that can be contained in the item. If the digits specification contains any decimal (fraction) places, CA Ideal makes sure that the numeric value is aligned properly (according to the digits specification), but without the implied decimal point. For example, if the digits specification is 3.2 and the value is 123.45 in packed decimal format, the internal value is X'12345C'.

For *zoned decimal* data items, the number of bytes needed to store the item is exactly the same as the number of decimal (base-10) digits. The sign is stored in the high-order four bits of the last byte. Thus, data item ZONED-4, defined on the following sample panel, requires four bytes of storage.

```
=>


---------------------------------------- -----
IDEAL: PARAMETER DEFINITION  PGM DEMO$01 (001) TEST  SYS: DEM  FILL-IN

COMMAND LEVEL FIELD NAME         T I CH/DG OCCUR  U M  COMMENTS/DEP ON/COPY
------  ----- ------------------- - - ----- -----
======  ===== ====== T O P ====== = = ===== =====  = =  =======================
......  1     ZONED-4            N Z     4        U I  :COBOL: S9(4) DISPLAY
......                                                 :  ASM: DS    ZL4
......                                                 :Format: |Fd|Fd|Fd|sd|
......
......                                                 : d=digit (X'0'-X'9')
......                                                 : s=SIGN  (X'A'-X'F')
......                                                 : F=X'F'
```

For *packed decimal* data items, the number of bytes needed to store the item is half the number of decimal (base-10) digits, except that one half-byte (the low-order four bits of the last byte) must be reserved for the sign. Thus, for the data item defined as follows:

```
=>
-----------------------------------------------
IDEAL: PARAMETER DEFINITION  PGM DEMO$01 (001) TEST          SYS: DEM  FILL-IN

COMMAND LEVEL FIELD NAME         T I CH/DG OCCUR  U M  COMMENTS/DEP ON/COPY
------  ----- ------------------- - - ----- -----
 ====== ===== ====== T O P ====== = = ===== =====  = =  =======================
000100  1     PACKED-5           N P     5        U I  :COBOL: S9(5) COMP-3
000200                                                 :  ASM: DS    PL3
000300                                                 :Format: |dd|dd|ds|
000400
000500  1     PACKED-4           N P     4        U I  :COBOL: S9(4) COMP-3
000600                                                 :  ASM: DS    PL3
000700                                                 :Format:  |0d|dd|ds|
000800
000900                                                 : d=digit (X'0'-X'9')
001000                                                 : s=SIGN  (X'A'-X'F')
001100                                                 : 0=X'0'
```

The field named PACKED-5 requires three bytes for storage (five half-bytes-one for each digit, plus one half-byte for the sign, equals six half-bytes or three bytes). The field named PACKED-4 also requires three bytes for storage because: A four-digit packed numeric field actually can be contained in two and one-half bytes. Since fields are always allocated in whole bytes, this value is rounded up to three bytes. This means that there is an unused half-byte (the high-order four bits of the first byte) that could contain a decimal digit. In fact, CA Ideal returns a run-time error if this position ever exceeds zero.

Binary fields present a special problem. CA Ideal supports only two types of binary fields: Half word (two bytes) and full word (four bytes). CA Ideal (like COBOL) still uses the convention that the field size is represented in number of decimal (base-10) digits. However, since binary numbers are base-2, there is no simple conversion from number of binary bytes to number of decimal (base-10) digits.

For example, the largest number that can be contained in a binary half word is 0111111111111111 (base-2), or 32,767 (base-10). All four-digit (base-10) numbers (up to 9,999) can thus be represented in a binary half word, but not all five-digit (base-10) numbers can. Specifically, the five-digit (base-10) numbers 32,768 through 99,999 cannot be represented in a binary half word.

Therefore, the algorithm for CA Ideal (and for COBOL) reasons as follows: If you specify four decimal (base-10) digits, a binary half word is sufficient, so two bytes (not aligned) are allocated. However, if you specify five decimal (base-10) digits, a half word is not sufficient for all five-digit (base-10) numbers, so the next largest size (a full word) is required. Therefore, four bytes (not aligned) are allocated.

Since the smallest machine format binary field size is a half word, two bytes (not aligned) are also allocated for fields specified with decimal (base-10) digits of 1, 2, and 3. This reasoning leads to the following algorithm: 1 to 4 decimal (base-10) digits require a binary half word (2 bytes) and 5 to 9 decimal (base-10) digits require a binary full word (4 bytes). Thus, the data item definition follows:

```
=>


-----------------------------------------
IDEAL: PARAMETER DEFINITION  PGM DEMO$01 (001) TEST          SYS: DEM  FILL-IN

COMMAND LEVEL FIELD NAME         T I CH/DG OCCUR  U M  COMMENTS/DEP ON/COPY
 ------  ----- ------------------ - - ----- -----  - -  ----------------------
======  ===== ====== T O P ====== = = ===== =====  = =  =======================
000100  1     BINARY_1           N B     1        U I  :COBOL: S9(1) COMP
000200                                                 :  ASM: DS    XL2
000300
000500  1     BINARY_4           N B     4        U I  :COBOL: S9(4) COMP
000600                                                 :  ASM: DS    XL2
000800
000900  1     BINARY_5           N B     5        U I  :COBOL: S9(5) COMP
001000                                                 :  ASM  DS    XL4
001100
......  1     BINARY_9           N B     9        U I  :COBOL: S9(9) COMP
......                                                 :  ASM  DS    XL4
```

The fields named BINARY_1 and BINARY_4 each are allocated two bytes (not aligned) and the fields named BINARY_5 and BINARY_9 each are allocated four bytes (not aligned).

To be flexible with other language conventions, CA Ideal does not align binary operands in data item definitions.

CA Ideal does not support binary numbers larger than nine decimal (base-10) digits. If it is necessary to pass a larger number to a non-ideal subprogram, it must be passed as a zoned decimal or packed decimal field.

CA Ideal does not support binary fields other than half word (two bytes) and full word (four bytes). If such a non-standard binary field is contained in a dataview created for a CA Datacom/DB table, CA Ideal flags the field with a warning message and treats the field as alphanumeric. This warning message allows the program to run and compile. If it is necessary to pass a binary dataview field other than two or four bytes, the field must be passed as an alphanumeric field and handled appropriately by the subprogram.

Equivalent specifications for the CA Ideal digits column (number of decimal or base-10 digits) and number of bytes of storage for each internal type are summarized in the following table.

	DG Column in CA Ideal (Number of Decimal or Base-10 Digits)	**Number of Bytes Storage**
Zoned decimal	n	*n*
Packed decimal	n	(*n*+1)/2 (round .5 up to next whole integer)
Binary	1-4	2 bytes (not aligned)
	5-9	4 bytes (not aligned)

At run time, unlike COBOL, CA Ideal enforces the limit on the number of decimal digits specified in the DG column, regardless of the number of bytes of storage.

Panel fields are zoned.

To calculate the internal storage required for nullable fields, add two to the number of digits.

Expression

An expression is a set of one or more related items that can be reduced to a single value. Expressions can be alphanumeric, arithmetic, or numeric.

Alphanumeric expression

An expression consists of alphanumeric literal, elementary alphanumeric field, variable length field, alpha group, or alphanumeric function.

Arithmetic expression

An arithmetic expression is a series of arithmetic operands and operators that can be reduced to a single numeric value. An operand can be any numeric expression. Arithmetic operators include plus (+), minus (-), times (*), divided by (/), and exponentiation (**exponent).

The *exponent* is a single numeric field or literal with a positive integer value or an alphanumeric field that consists only of numerals; an exponent cannot be an arithmetic expression; the maximum value for an exponent is 999.

An arithmetic expression that contains one or more null values results in a null value.

Parentheses specify the order of evaluation of an arithmetic expression. But if you omit parentheses, exponentiation is performed first, multiplication and division are performed next, and addition and subtraction are performed last. When two or more operators appear without parentheses and are at the same level, evaluation is from left to right. For example, if numerals are assigned to the following field,

```
A = 7   C = 8
B = 3   D = 2
```

Then the following expression is equivalent to,

```
A + B - C/D  equals  6
```

whereas the following expression with parentheses is equivalent to,

```
(A + B - C)/D  equals  1
```

In an arithmetic expression, a "-" or "+" must be surrounded by blanks or parentheses as follows:

```
A + B*C - D

(A*B)-(C*D)
```

Up to 12 levels of parenthetic and function nesting are permitted in arithmetic expressions.

You cannot use non-alpha groups, condition names, flags, and alphanumeric literals in arithmetic expressions.

The following are examples of valid arithmetic expressions:

```
A + B
A*B
(A*B)/C
(A + B**C)/D**E
```

Numeric expression

A numeric expression is a numeric literal, numeric field, date field, numeric function, or arithmetic expression. You can convert alphanumeric fields or alpha groups to numeric using the $NUMBER function in place of numeric expressions, but a compile-time warning is issued.

Field

A field is the smallest named unit of data that a program can access.

Alphanumeric field

An alphanumeric field is a field with a data type of X. The maximum length is 32,000 characters.

Date field

A field with a data type of D. The date field has a numeric value indicating an integer number of days from December 31, 1900 (day zero), plus or minus. You can use date fields anywhere numeric fields can be used, except where noted.

You cannot transmit date fields to a panel. To use a date field in a panel display, first use the $DATE or $STRING function to convert the date field to an alphanumeric field, and then transmit it. To use a date with a positive value as a key, store it in an unsigned numeric field.

CA Ideal processes dates between 2000 BC and 9999 AD. Any other date causes a runtime error.

SQL DATE, TIME, and TIMESTAMP fields are supported for CA Datacom/DB native access; however, they are converted to character (Type X) with the following lengths:

```
DATE  10
TIME  8
TIMESTAMP  26
```

These fields are stored as binary, unsigned, integer fields. CA Ideal automatically handles conversion between the storage format and the display format (character). In addition, when a record is added with a FOR NEW statement, any DATE, TIME, or TIMESTAMP fields are initialized using the system date and time.

Flag

A field with a type of F has a value of TRUE or FALSE. You can use flags in SET or MOVE statements where they are assigned the value TRUE or FALSE (see SET or MOVE Format 3) as all or part of a condition or in a LIST statement where they appear as the values T or F (see the description of the LIST Statement). Flags take one byte of storage and contain a T or an F.

Group

Group is a named logical collection of one or more fields or groups, panels, and data views.

Alpha Group

An alpha group is a group in which all subordinate fields are either alphanumeric fields, redefinitions of alphanumeric fields, or alpha groups. To qualify as an alpha group, the group must be in a dataview, in working data, or in an identical-match parameter group.

For purposes of syntax, you can use an alpha group interchangeably with an alphanumeric field.

NonAlpha Group

A non-alpha group is a group that is not of alpha mode. You can use all non-alpha groups in the following contexts:

- In SET/MOVE … BY POSITION/NAME statements.
- As dynamic match parameters on CALL statements.
- As operands of some alphanumeric functions.

Restrictions on nonalpha groups

The use of some non-alpha groups is restricted because they contain internal, non-displaying fields CA Ideal uses. The restricted groups are:

- Groups that were passed as dynamic parameters
- Groups containing subordinate variable length fields
- Groups containing subordinate nullable fields
- Panel groups
- Groups containing SQL DATE, TIME and TIMESTAMP fields

You cannot use restricted groups in the following contexts, although other non-alpha groups can:

- As source operands in SET/MOVE statements that move a value to an elementary numeric or alphanumeric field.
- As operands in simple relations in conditional expressions.
- As identical match parameters on CALL statements.

When you use non-restricted groups in these contexts, the non-alpha group is treated as an alphanumeric field whose length is the size of the group. Subordinate fields are not converted. The hexadecimal contents are moved or compared.

Nullable field

A nullable field is a field defined as eligible to receive null values. The initial value, if not specified, is the null value. Fields in working data, parameter data, panels, and dataviews defined through SQL can be nullable.

The null value is an unknown value. CA Ideal maintains a nullable field as a single field with a null value indicator.

You can assign a null value only to a null-eligible field.

You cannot redefine nullable fields or groups that contain subordinate nullable fields. Groups that contain subordinate nullable fields are restricted

Null values display as question marks (?) in reports.

Panel fields with null values are treated as empty ($EMPTY is true). Null values display as question marks (?) in panels.

You can use nullable fields in built-in functions, except as the arguments for keyword parameters, such as "START=n" in $SUBSTR.

Numeric field

A numeric field is a field with a type of N (numeric) or U (unsigned numeric) and a numeric value. The maximum length of the value of a numeric field in CA Ideal is 31 digits. For more information, see the Numeric literal section in this chapter.

Subordinate field

A subordinate field is a field that is in a group or alpha group.

Variable length field

A variable length field is a field with a type of V. The size of the field depends on the value, up to the specified characters/digits length. You can use variable length fields anywhere you can use alphanumeric fields, except as noted. The maximum length of a variable length field is 32,000 characters.

You cannot redefine variable length fields or groups containing subordinate variable length fields. Groups that contain subordinate variable length fields are restricted non-alpha groups. For more information, see the Non-Alpha Group section in this chapter.

Functions

Functions are requests that return values for various common services. PDL functions are documented in the "Symbolic Debugger Commands" chapter.

All PDL function names start with a dollar sign, for example, $NUMBER. You invoke functions by coding the function name at the point in the program where the value is needed. Any parameters required are enclosed in parentheses.

The four types of built-in functions are numeric, alphanumeric, Boolean, and pseudo functions. "Types of Operands for PDL Functions" appendix contains a listing of the PDL built-in functions by type and shows what types of operands can be used with each function.

Numeric function

A function that returns a numeric value. An example of a numeric function is $COUNT. You can next numeric functions in other numeric functions to any level.

Alphanumeric function

A function that returns an alphanumeric value. An example of an alphanumeric function is $STRING. You can nest alphanumeric functions in other alphanumeric functions to three levels. In the following example, the number above each function represents the level of nesting:

```
          1       2                  3
$STRING($SUBSTR($STRING('XXXXBEGIN',$SUBSTR(FLD,START=4))))
```

Boolean function

A function evaluates to True, False or Unknown. For further information refer to the description of Boolean function earlier in this chapter. An example of a Boolean function is $ALPHABETIC.

pseudofunction

A request similar to a function; used in SET statements to assign values to certain system variables, for example, SET $FINAL-ID = 'PAY'

Identifier

The name of a group or field defined in working data or parameter data that, optionally, is qualified and subscripted. This chapter contains sections on names, qualified names and subscripted names.

Restrictions

All level-1 identifiers (dataviews, panels, or the highest level (level-1) of working data or parameter data) must be unique in each program.

Identifiers must be unique in dataviews, panels, or the highest level (level-1) of working data or parameter data.

You can use reserved words as identifiers at level 2 or lower. They must be qualified regardless of whether they appear more than once. For example, a field named with the reserved word NAME in a dataview named EMPLOYEE must always be identified with the qualified name EMPLOYEE.NAME.

You can use SQL reserved words as identifiers in PDL (but not in embedded SQL). You can use PDL reserved words as identifiers in SQL. However, neither of these is recommended.

Examples of Valid Identifiers

The following names are all valid identifiers. The second, third, and fourth names are qualified. The last two names are subscripted.

```
PANEL1
PAYROLL.NAME
EMPLOYEE.ADDRESS
PAYROLL.ADDRESS MONTH(7)
DAY(COUNT+1)
```

Literal

A sequence of symbols whose value is implicit in the characters themselves. Every literal must be contained entirely on one line. PDL uses the following types of literals:

Numeric literal

Any series of one to 31 digits, with one optional decimal point, and no embedded blanks, optionally preceded by a sign ("" or "-"). The following are examples of valid numeric literals:

```
5
22.3
-16
-17.3
+92
1745375
```

Alphanumeric literal

Any series of characters including blanks, surrounded with delimiters. The delimiters can be double quotes (") or apostrophes ('). The starting and ending delimiter must be the same character. Some examples of valid alphanumeric literals are:

```
"12345"
'CITY'
"REENTER CODE"
'STATE'
'COUNTY #124-A***'
"***ERROR***"
"O'REILLY, SCHWARTZ, & SMITH ASSOCIATES"
```

Boolean literal

The value is TRUE or FALSE. Do not enclose Boolean literals in delimiters. They define the initial value of flags and set the value of flags.

Name

Every dataview definition, report definition, panel definition, and program definition CA Ideal uses and every field or group defined in working data or parameter data must have a name.

Rules for Valid Names

Must be an alphanumeric string.

- Begins with a letter (A through Z) or national character ($, @, #).
- Consists only of numerals, letters, national characters, embedded underlines or hyphens.
- The last character cannot be a hyphen or underline.

Maximum Name Lengths

- Field or group - 32 characters
- Dataview definition for CA Datacom/DB native command access or sequential file - 18 characters
- Dataview definition for SQL access - 27 characters
- Object name - 18 characters
- Period - 1 character
- Authorization (or creator) ID - 8 characters
- Program definition - 8 characters
- Panel definition - 8 characters
- Report definition - 8 characters

Assignment Name Restrictions

The following restrictions are placed on the assignment of names:

- Dataview definition names must be unique across CA Ideal.

  For SQL dataviews, the fully qualified name (that is, auth_id.obj_name) must be unique across CA Ideal. The object name without qualification must be unique in each program.

  For example, the SQL dataviews ID.PAYROLL and SBL.PAYROLL can exist in separate program resource tables, but not in the same program resource table. Similarly, you cannot specify the CA Datacom/DB native command access dataview PAYROLL and the SQL dataview ID.PAYROLL in the same program resource table.

- Panel definition names, program definition names, and report definition names must be unique across CA Ideal systems.
- You can use reserved words as the names of fields or groups at level 2 or lower if they are qualified (see Qualified Name).

Qualified Name

If a name appears in more than one dataview, panel, or level-1 working or parameter data item, it must be qualified. Qualification ensures that a group or field is uniquely identified, even if the same name appears in another dataview, panel, or level-1 data item. The same rule applies to data names put into working or parameter data using a COPY DVW clause.

The name is qualified by prefixing it with the appropriate dataview name, panel name, working data level-1 name, or parameter data level-1 name, and a period. For example, if fields named STATUS appear in both a dataview named EMPLOYEE and a dataview named PAYROLL, the fields must be identified as EMPLOYEE.STATUS and PAYROLL.STATUS.

Subscripted name

An identifier can include subscripts. You can include one to three subscripts after the field or group name, separated by commas and enclosed in parentheses. If occurring items are nested in other occurring items, the number of subscripts must equal the number of levels of nesting. The definition of the field or group in working data must include the number of occurrences.

Thus, an identifier consists of a name, optionally preceded by a qualifier, and optionally followed by one to three subscripts. The syntax for the identifier is as follows:

```
[dataview-name.]
[panel-name.   ] name [(sub[,sub]])]
[level-1-name. ]
```

sub A subscript of the form:

```
{name           } [ {+ } {name           }]
{numeric literal} [ {- } {numeric literal }]
```

name-The name of a numeric field containing a valid integer numeric value. The field cannot be null eligible. Names used in subscripts can be qualified, but cannot be subscripted.

numeric literal-An integer numeric value with no decimal places.

Note: The value of a subscript must be an integer between one and the number of occurrences of the name being subscripted.

Subscripts are defined in the program definition working data. For details, see the *Creating Programs Guide*.

Examples of valid names

The following identifiers are valid names:

```
PANEL1              $156_@
NEW_ENTRY           #12345
PAYROLL-PROC        X_627B
PAYROLL-DVW         SQLID1.CUSTOMER
```

Note: You can reference panel names and dataview names as level-1 group names. Creating the panel or dataview defines the name. You do not have to define it in working data.

NULL

The null value. This keyword defines the initial values of null-eligible fields and sets null-eligible fields to this value. It is also used in the null condition.

Parameter

A data item used in a CALL statement or a RUN command.

Procedure

A named, functional collection of statements. You can use procedures to divide a program or subprogram into logical subcomponents.

Reserved word

A word with a special meaning to PDL. You can use reserved words as identifiers if they are qualified (see the section Qualified Names in this chapter**.**). "PDL Reserved Words" appendix contains a list of PDL reserved words. "PDL Reserved Words" appendix contains a list of SQL reserved words.

Statements

Simple directives in the PDL language, like SET, MOVE, and TRANSMIT, or constructs, such as the IF and FOR constructs. For purposes of syntax notation, the term *statement* also includes the absence of a PDL directive in a place where a statement is an optional part of a construct. For example, the following syntax statements could be satisfied by the IF….ELSEIF constructs:

```
IF condition
   statement-1
ELSE
   statement-2
ENDIF

IF MORE_RECORDS
ELSE
   DO ERROR_EMPTY
ENDIF
```

In this case, statement-1 is not specified and statement-2 is DO ERROR_EMPTY.

Subprogram

Any program called by another program. You must specify a subprogram as a resource of a calling program. For a description about how to specify program resources and an explanation of how subprograms are called by CA Ideal programs, see the *Creating Programs Guide*.

PDL Format Rules

There are very few rules for the formatting of a CA Ideal program. With the exception of the limitations outlined in Lexical Rules section, you can enter PDL programs in free format. You can continue statements over lines without the use of continuation characters, statements can begin in any column, and there are no spacing rules.

Different rules apply to embedded SQL (see the Format Rules section in the chapter "Procedure Definition Language Statements").

Lexical Rules

- The following are valid characters used as delimiters for identifiers, reserved words, and numeric literals: space, comma, left parenthesis (, right parenthesis), less than <, greater than >, equals =, asterisk *, slash /, not ¬, ampersand &, vertical bar |, and colon :.
- You can use apostrophes (') and quotation marks (") interchangeably to delimit alphanumeric literals, but the leading and final delimiter must match.
- Identifiers, literals, labels, and reserved words can appear in the right-most and left-most columns of the source record.
- Anything to the right of a colon (:) or double hyphens (--) is treated as a comment.
- You cannot break words from one line to the next.
- You can leave lines blank.
- See the definitions of names, identifiers, literals, functions, and so on earlier in this chapter for further information about permitted use.
- Labels of procedures, FOR constructs, and loops take the form:

  <<*name*>>

EJECT Statement

The EJECT statement causes the compilation listing to skip to the top of a page. It must be on a line by itself.

Comment

A comment is a character string that serves as documentation in a program. Comments are not executable. Any line in a PDL program can contain a comment. A line that begins with : or -- is treated as a comment.

```
:      text of comment
--     text of comment
```

: (colon) All characters to the right are treated as a comment. You cannot use the colon for comments in embedded SQL.

-- (double hyphens) All characters to the right are treated as a comment.

Because the comment ends at the end of the line, no special character is required to terminate the comment.

Example

```
: this is regarded as a comment.
SET A = B + C  :  this is regarded as a comment,
SET D = A + 1  -- and this, too, but not the SETs
```

Converting Between Numeric and Alphanumeric

When you use a numeric field in an alphabetic context, it is converted to a display form. See $STRING, $EDIT, LIST, MOVE, and SET.

When you use an alphanumeric field in a numeric context, it is converted to numeric form if possible before it is evaluated. See the $NUMBER function.

If an alphanumeric value is converted to a numeric value, a compile-time warning message is issued and, if the source does not contain a valid numeric value, a runtime error occurs.

Data Definition Conventions

CA Ideal application programs use data from a variety of sources. This data includes:

- Working data
- Parameter data
- The logical structures of a relational database as defined through dataview definitions
- The record structures of a sequential file as defined through dataview definitions
- The record structures of a VSAM file as defined through dataview definitions

When CA Ideal uses any of this data, it is defined in similar ways.

The *Creating Programs Guide* contains explanations of how working data for CA Ideal programs and parameter data for both CA Ideal and non-ideal subprograms is defined. The structures used in these explanations closely resemble the way in which dataviews appear to CA Ideal users.

For information about how CA Ideal applications use dataviews, see the FOR Statement in this guide. For more information about how CA Ideal uses dataviews, see the *Creating Dataviews Guide.*

Chapter 2: SQL Concepts and Language Elements

CA Ideal supports SQL access to CA Datacom/DB and DB2 in PDL programs. This chapter describes the SQL statements and search conditions CA Ideal supports.

You can code embedded SQL statements directly in a CA Ideal program's procedure section. SQL statements are delimited by the words EXEC SQL and END-EXEC. The features of SQL embedded in a CA Ideal application follow the rules and descriptions for embedded SQL in a COBOL environment, except where noted in this section. The EXEC SQL statement is described in the "Built-In Functions" chapter.

The CA Ideal editor lets you enter template commands in the margin of the program procedure fill-in. SQL template commands automatically generate syntactically correct SQL statements. SQL template commands are described in the *CA Ideal Command Reference Guide*.

You can also use the FOR construct to access databases using SQL. When you use the FOR construct with a dataview defined to access an SQL object, CA Ideal automatically generates optimized SQL statements to perform the same functions. For description of the FOR statement, see the "Built-In Functions" chapter.

You can print the generated SQL on the compiler listing. For more information about COMPILE command, see the *Command Reference Guide*.

SQL Dataviews

A CA Ideal dataview corresponds to a complete SQL table, view, or synonym. To create and use an SQL dataview, perform the following tasks:

1. Issue a CATALOG command with the name of the SQL object.

   The catalog process retrieves the definition of the object from the database. It creates a dataview entity-occurrence for the object in the dictionary facility and a dataview object module in the virtual library system.

2. Specify the dataview with the object's authorization ID in a program's resource fill-in.

Any embedded Data Manipulation Language (DML) statements or FOR constructs can access the cataloged object from that program.

For more information about how to define SQL dataviews, see the *Creating Dataviews Guide*.

CA Datacom/DB Access Plans

Every program that accesses CA Datacom/DB using embedded SQL or FOR constructs must have an access plan that is unique to that program. CA Ideal builds the plan for the application and binds each SQL statement as part of program compilation. The authorization ID and other plan options are taken from the program environment fill-in (described in the *Creating Programs Guide*).

You can change the plan options and rebind a plan without recompiling the program by using the ALTER PROGRAM ENVIRONMENT command, followed by the REBIND command. You can also define alternate plans for a program and select the appropriate plan at runtime. For more information about defining and maintaining alternate plans, see the *Administration Guide*.

DB2 Application Plans and Packages

Application plans and packages for DB2 are created in CA Ideal using the plan definition facility. For more information about this facility, see the *Administration* Guide. The binding of plans and packages is performed in batch CA Ideal, using the GENERATE PLAN and GENERATE PACKAGE commands.

You can execute programs against DB2 in dynamic or static mode, for both embedded and generated SQL, without any changes to the source code. If you generate and use an application plan, the application runs in static mode. You can also have the application switched back to dynamic mode automatically for quick maintenance.

SQL NULL Attribute

CA Ideal supports the NULL concept in SQL. Fields in dataviews that are defined as null-eligible do not require indicator variables. Rather, the programmer uses the keyword NULL, as in IF FIELD_X IS NULL. CA Ideal also handles the null attribute in conditional and arithmetic expressions and in working data, parameter data, and on panels and reports.

Error Processing

A CA Ideal program can access the SQL Communication Area (SQLCA) for the last SQL statement executed in a program or run-unit. You can define the SQLCA work area in the program's working data or parameter data or code $SQL functions in any program procedure or $ERROR functions in the Error procedure. The SQLCA is described further in the SQLCA section in this chapter.

CA Ideal also supports the SQL WHENEVER statement (described in the SQL Language Elements section in this chapter).

Because SQL implementations can differ, the CA Ideal compiler can issue warnings for possible semantic errors in user-generated SQL. Such statements are passed to the database management system for final determination.

Active Dictionary Facility

CA Ideal keeps track of the application model, which includes systems, programs, dataviews, panels, and reports. The dictionary facility is automatically populated as CA Ideal operates. Dictionary facility information verifies the integrity of developer actions.

Mixed SQL Sites

A mixed SQL site can access multiple databases using SQL. A single program can access multiple databases, but each dataview name (that is, each SQL object name) must be unique. Across a CA Ideal system, each fully-qualified object name must be unique.

With a mixed site, the database the SQL statements access depends on the type of statement:

- Each data manipulation (DML) statement, such as INSERT, requires a cataloged dataview for its object. The dataview specifies which database contains the object and processes the statement.
- The COMMIT, ROLLBACK, and WHENEVER statements affect all databases the application accesses.
- Statements such as GRANT and EXECUTE IMMEDIATE that do not access objects defined as CA Ideal dataviews can access only one database from a program. This database is called the primary database. The primary database processes supported SQL statements listed in the following two charts as *no dataview required*.

You can specify the primary database as a site or session default, and override it for individual programs. For more information, see the SET ENVIRONMENT SQL command in the *Command Reference Guide* and the program environment fill-in in the *Creating Programs Guide*.

Supported SQL Statements

CA Ideal PDL supports the SQL following statements. The CA Datacom SQL statements are listed first, followed by the DB2 statements.

CA Datacom/DB Database SQL Option Statements Supported in PDL:

Statement Type	Statement	Comments
DML Dataview required in program resource fill-in	CLOSE DECLARE CURSOR DELETE FETCH INSERT LOCK TABLE OPEN SELECT UPDATE	
DML No dataview required	COMMIT ROLLBACK WHENEVER	Operates like PDL CHECKPOINT. Operates like PDL BACKOUT. See the section titled WHENEVER Statement.
DCL No dataview required	COMMENT ON GRANT REVOKE	

DB2 SQL Statements Supported in PDL:

Statement Type	Statement	Comments

DDL
No dataview required
	CREATE SCHEMA
	CREATE SYNONYM
	CREATE TABLE
	CREATE VIEW
	DROP	

Statement Type	Statement	Comments
DML Dataview required in program resource fill-in	CLOSE DECLARE CURSOR DELETE FETCH INSERT LOCK TABLE OPEN SELECT UPDATE	 FOR FETCH ONLY supported OPTIMIZE FOR supported
DML No dataview required	COMMIT ROLLBACK WHENEVER	Operates like PDL CHECKPOINT. Operates like PDL BACKOUT. See the section titled WHENEVER Statement.
DCL No dataview required	Dynamically, using EXECUTE IMMEDIATE: ALTER COMMENT ON CREATE DROP	
DDL No dataview required	Dynamically, using EXECUTE IMMEDIATE: EXPLAIN GRANT LABEL ON REVOKE	

WHENEVER Statement

The SQL WHENEVER statement embedded in a PDL program specifies the action to take when a specified condition occurs during the execution of a subsequent embedded SQL statement. The action specified for a given condition applies to all SQL statements that follow in listing sequence until another WHENEVER statement for that condition overrides it.

This statement has the following format:

```
                         [CONTINUE           ]
            {NOT FOUND }  [DO ERROR           ]
WHENEVER    {SQLERROR  }  [DO procedure       ]
            {SQLWARNING}  [PROCESS-NEXT-label ]
                         [QUIT-label         ]
```

NOT FOUND| SQLERROR| SQLWARNING

The type of SQL exception condition.

CONTINUE

Default: Specifies that execution continues with the next sequential statement in the program.

DO ERROR

Invokes the CA Ideal error procedure (user-specified or default) from the embedded SQL statements. $ERROR function values are available in the error procedure.

DO procedure

Executes the procedure identified by the specified procedure label and returns control to the statement following the END-EXEC.

PROCESS-NEXT-label

A PROCESS NEXT statement specifying that the current iteration of the current LOOP or FOR EACH construct terminates. See the PROCESS NEXT statement in the next chapter.

QUIT-label

A QUIT statement specifying execution of a PDL QUIT with the specified option. See the QUIT statement in the next chapter.

If no WHENEVER statement is included in the program for a given condition, the default is CONTINUE.

The PDL $ERROR functions, error procedure, and the command LIST ERROR do not apply to errors encountered in processing embedded SQL statements.

PDL FOR constructs, even though they generate SQL requests, are bound by PDL error handling rules, not WHENEVER specifications.

For a mixed SQL site, the WHENEVER applies to SQL statements accessing any database. You can determine the database management system accessed by the last SQL statement executed by using the $SQL-DBMS function.

Example

In this example, if the following conditions are encountered in any embedded SQL after the WHENEVER in listing sequence, the indicated action occurs.

```
EXEC SQL WHENEVER SQLERROR DO ERRPROC END-EXEC
EXEC SQL WHENEVER NOT FOUND PROCESS NEXT MAIN-LOOP END-EXEC
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC
```

Condition	Action
SQLERROR	The procedure ERRPROC is executed.
NOT FOUND	The next iteration of MAIN-LOOP is processed.
SQLWARNING	Processing continues with the statement following the statement that caused the condition.

Extension to INTO and VALUES Clauses

CA Ideal PDL supports the following extension to the SQL INTO and VALUES clauses.

You can move data into PDL group data items using FETCH and SELECT statements and from PDL group data items to tables or views using the INSERT statement through the BY POSITION option.

The extension has the following format:

```
INTO host-structure [BY POSITION]
VALUES host-structure [BY POSITION]
```

host-structure

A PDL group identifier.

BY POSITION

Moves data between each elementary field in the PDL group and each column in the currently accessed row. The structures must be compatible as defined for the BY POSITION option of the PDL MOVE statement. That is, you cannot define a group referenced in an INTO clause with an OCCURs attribute, nor can you redefine any of its subordinate fields. These groups can, however, contain subordinate groups. The data types must be compatible and conform to SQL requirements.

If you do not specify BY POSITION, you cannot specify a non-alpha group. An alpha group is treated as an elementary alpha field.

Also, you can use the COPY DATAVIEW clause with a Working Data fill-in to automatically create an image of an SQL dataview in a group item.

DB2 SQL Not Supported

The CA Ideal PDL does not support the following DB2 SQL statements and clauses.

- SQL clauses used exclusively to process SQL statements dynamically. They are the USING clause and prepared statement-name reference.
- The SQL statements that serve as documentation and that support extended semantic checking by the IBM precompiler. Their functions are already performed in CA Ideal. These statements are:
  - DECLARE TABLE
  - DECLARE STATEMENT

  The interactive SQL statements and most of the SQL statements that support dynamic SQL. These statements are:
  - DESCRIBE
  - EXECUTE
  - PREPARE
  - Interactive SELECT

SQLCA

The SQLCA is a work area that retrieves information about the last SQL statement processed, either embedded SQL or SQL generated by a FOR construct. You can access SQLCA fields in CA Ideal programs using the following functions:

- A series of $SQL built-in functions. The $SQL built-in functions return information about the last SQL statement processed in the application or run unit (that is, in the program and its subprograms). The data type the function returns is the same as the type of the associated field. For more information about $SQL functions, see the "Symbolic Debugger Commands" chapter.
- The $ERROR-DVW-STATUS function to return the SQLCODE. The $ERROR functions return information about the last SQL statement generated by a FOR construct in an Error Procedure or WHEN ERROR clause.

- A copy of the SQLCA in working data or parameter data. You can have a copy for each database with SQL access. Each consists of a single level-1 group item for the SQLCA and subordinate items for the individual SQLCA fields. The SQLCA fields return information about the last SQL statement in the program (not subprograms) that the database processed. For example, if you have SQLCAs for DB2 and CA Datacom SQL access, the DB2 SQLCA only returns information about the last SQL statement DB2 processed. For more information, see the COPY SQLCA clause under Working Data or Parameter Data in the *Creating Programs Guide*.

In a non-CICS environment, if a PDL CHECKPOINT or BACKOUT statement is executed and the database was accessed, then CA Ideal executes a SQL COMMIT or ROLLBACK, changing the contents of the SQLCA.

Supported SQL Language Elements

Condition

CA Ideal supports SQL basic predicates and standard SQL relational operators in embedded SQL and FOR constructs with SQL dataviews.

The following SQL relational predicates are supported:

- ALL
- ANY
- BETWEEN
- EXISTS
- IN
- LIKE
- NULL
- SOME

In addition, you can use the following PDL relational operators in embedded SQL predicates:

Relational Operators	Symbol
EQUAL	EQ
NOT EQUAL	NE
GREATER THAN	GT
NOT GREATER THAN	LE

Relational Operators	Symbol
LESS THAN	LT
NOT LESS THAN	GE

The symbol ¬ can replace NOT. THAN is optional.

You can combine conditional expressions using AND and OR. You can use the characters & and | for AND and OR, respectively.

PDL implied subjects and implied operators do not apply in embedded SQL conditions.

Data Types

CA Ideal supports the following SQL data types:

CA Datacom SQL	DB2
Character	Fixed-length string
Variable length string	Variable length string
Small integer	Small integer
Large integer	Large integer
Numeric	
Decimal	Decimal
Date	Date
Time	Time
Timestamp	Timestamp

All data types include the null value.

CA Ideal fields that correspond to SQL date and time type columns appear to PDL with the following attributes, unless DB2 user exits override the defaults:

Date Columns: X(10)

Time Columns: X(8)

Timestamp Columns: X(26)

For CA Datacom SQL access, variable length character strings are shown as type X, with the maximum length as the length.

Not Supported: Graphic string and any type of floating point.

For more information, see the *CA Datacom/DB Database and System Administration Guide*.

Function

All SQL functions are supported in embedded SQL. You can use SQL functions in the search condition of a FOR construct with an SQL dataview where the functions are allowed by SQL rules.

PDL functions are prohibited in embedded SQL and FOR constructs with SQL dataviews.

Host Variables

A CA Ideal group, field, or parameter that is defined in a dataview, panel, working data, or parameter data, and that is used in embedded SQL.

You can prefix a host variable in embedded SQL with a colon; that is:

```
:host-identifier
```

The identifier must immediately follow the leading colon.

For DB2: You can omit the colon from a host identifier when the host identifier is not a reserved word or when it meets any of the shown in the list that follows.

For CA Datacom/DB: You can omit the colon from a host identifier when each host identifier used in a statement meets the following conditions:

- It is qualified by a host structure (group) name that is not a reserved word.

  For example, you can specify the subsidiary field SUR_NAME, which is part of group LONG_NAME, as :SUR_NAME, as LONG_NAME.SUR_NAME, or as :LONG_NAME.SUR_NAME.

- It is referenced in a context where column names are illegal (that is, in an INTO or VALUE clause or in the LIKE or IN predicates) and it is not an SQL reserved word or PDL verb.

You can use an SQL column name defined in a CA Ideal dataview as a host variable, but only in the logical scope of a FOR statement executed for the dataview. See the FOR Statement for SQL in The chapter Built-In Functions for a description of embedded SQL and the FOR.

An alpha group is treated as an elementary alpha field in embedded SQL except when it is used in an INTO or VALUE clause with BY POSITION. In that case, each subfield in the group is treated as an elementary alpha field. You can use a non-alpha group as a host variable only with BY POSITION. In that case, each subfield in the group is treated as an elementary field of the appropriate type.

You can use COPY DATAVIEW to automatically include an image of an SQL dataview in a working data or parameter data group item.

Host variables that correspond to SQL date and time type columns can be alphanumeric fields that follow the default formats (listed previously under data types) or that follow local date and time default formats specified in a DB2 user exit. You can also use CA Ideal date fields as host variables that correspond to SQL date type columns. You can use a CA Ideal date field anywhere an SQL date type column is acceptable, except as the target of a FETCH INTO when a local default date exit is in effect.

Indicator Variables

You cannot specify explicit indicator variables. CA Ideal manages a null value indicator to handle null eligible host variables. To access a column that can have null values, use a host variable defined as null eligible (see the section on nullable field in the "SQL Concepts and Language Elements" chapter). A non-ideal subprogram must provide explicit indicator variables if it is passed nullable fields.

You cannot assign a null value to a host variable that is not eligible to receive nulls because it will result in a run-time error.

Qualified Host Variable Identifiers

A host variable identifier must be qualified by a group name if it is not unique in the program.

For CA Datacom/DB: You cannot use a SQL reserved word as a host identifier.

For DB2: You can use a SQL reserved word as a host variable name if it is qualified by a group name.

You can omit the leading colon from a qualified host identifier. If you omit the colon and the group-name qualifier is the same as a table, view, or correlation name in the current SQL statement, the reference is assumed to be to a column, not to a host variable.

Identifiers

The SQL ordinary identifier-a letter followed by 0 to 17 characters-is supported in embedded SQL.

CA Ideal does *not* support delimited identifiers.

Literals

CA Ideal supports the following types of SQL literals in embedded SQL. FOR construct search conditions follow the rules for PDL literals.

- Integer constants (for example, +100, 64, -15)
- Decimal constants (for example, 25.4, -56.0, 99.0)
- Character strings, delimited by apostrophes (for example, 'literal') or quotation marks (for example, "literal").
- To embed an apostrophe (') in a character string, use quotation marks to delimit the character string (for example, "literal's"). To embed quotation marks in a character string, use apostrophes to delimit the string.

  In DB2: Hex literals, specified as X followed by a character string (for example, X'FFFF').

Name Conventions

The names of SQL objects and columns in embedded SQL follow the naming conventions of the appropriate database management system. CA Ideal supports the use of the tables, views, and synonyms in CA Datacom/DB and tables and views in DB2.

Every object included in an embedded DML statement or in a FOR construct must correspond to a CA Ideal dataview that is specified in the resource table of every program that uses that object.

Qualified Table and View Names

You can qualify the names of SQL objects with authorization IDs of up to eight characters. For example, you can qualify the table PAYROLL with the authorization ID HOU:

```
HOU.PAYROLL
```

The dataviews corresponding to SQL objects must have authorization IDs. You can use the dataview authorization ID to qualify the object name in the SQL presented to the database or you can code an authorization ID explicitly in the embedded SQL. In each program resource table that includes a dataview for an SQL object, you must perform the following tasks:

- Specify a dataview authorization ID. This authorization ID identifies the cataloged dataview corresponding to the object.
- Set the Q (qualifier) column for the dataview to either Y or N. This tells CA Ideal to use the dataview authorization ID in the SQL when the object name is not qualified in the embedded SQL or allow the database to supply its default authorization ID.

You have several options for qualifying an object name based on how its dataview is defined:

- You can always use the authorization ID specified in the program resource fill-in. Specify Y in the Q (Qualifier) column. Qualifying the name in embedded SQL is allowed, but not necessary.
- You can have CA Datacom/DB supply its default authorization ID. Specify N in the Q (Qualifier) column and do not qualify the name in embedded SQL.
- You can override the default for selected SQL statements by specifying N in the Q (Qualifier) column and, in those statements, qualifying the object name with the authorization ID from the program resource fill-in.

  This lets you access multiple objects with the same name but different authorization IDs using one CA Ideal dataview. Their structures must be compatible. For more information about defining a program, see the *Creating Programs Guide.*

Example

Consider the following program resource table definition:

```
Dataview             Auth-id   Q?

EMPLOYEE             SBL       Y
PAYROLL              HOU       N
```

Example

Consider also the following embedded SQL statements:

```
EXEC SQL                                 EXEC SQL
    SELECT                                SELECT ...
     FROM EMPLOYEE, PAYROLL                 FROM SBL.EMPLOYEE, HOU.PAYROLL
END-EXEC                                 END-EXEC

               EXEC SQL
                  INSERT INTO PAYROLL
                    (SELECT * FROM HOU.PAYROLL ...)
               END-EXEC
```

Example

CA Ideal generates the following clauses:

```
SELECT ...                               SELECT ...
  FROM SBL.EMPLOYEE, PAYROLL               FROM SBL.EMPLOYEE, HOU.PAYROLL


           INSERT ... INTO PAYROLL
                        (SELECT * FROM HOU.PAYROLL ...)
```

In the first example, the database qualifies the table name PAYROLL with an authorization ID.

In the second example, both table names are qualified with the dataview authorization ID from the program resource fill-in.

In the third example, two tables, HOU.PAYROLL and *xxx*.PAYROLL (where *xxx* is the CA Datacom/DB default authorization ID and is not HOU) are accessed using the one PAYROLL dataview specified in the program resource fill-in.

The ASSIGN AUTHORIZATION command lets you specify an authorization ID to use for all tables or views specified as unqualified in the resource table.

For DB2: The ASSIGN AUTHORIZATION command also lets you replace the dataview authorization ID specified in the resource table with a new authorization ID or generate an unqualified table or view name. The interactions of the resource table, embedded SQL, and the ASSIGN AUTHORIZATION command are explained in the *Command Reference Guide* and the *Administration Guide*.

For CA Datacom SQL access: The ASSIGN AUTHORIZATION command lets you select an alternate access plan at runtime.

Qualified Column Names

In embedded SQL, you can qualify a column name with the name of an SQL object. CA Ideal supports up to two levels of qualification. This means that you cannot qualify a column name with an object name that is itself qualified by an authorization ID. For example, column name EMP_NAME can be qualified by the table name EMP_TABLE, but it cannot be further qualified as in AUTHID.EMP_TABLE.EMP_NAME.

```
EMP_TABLE.EMP_NAME
```

You can achieve the same result by using correlation names or by letting CA Ideal qualify the object name using one of the methods described previously.

CA Ideal validates column references. It checks that:

- The column name is defined in the specified dataview.
- Null eligible variables select NULL eligible columns.
- Host variables are of compatible type for the columns selected or compared.

Reserved Words

For CA Datacom/DB native access, you cannot use a SQL reserved word as an identifier. For DB2, you can use a qualified SQL reserved word as an identifier.

For more information about reserved words, see the "*SQL Reserved Words*" appendix.

SQL Formatting Rules

A CA Ideal program has only a few formatting rules. With the exception of the limitations outlined as follows, you can embed SQL statements in free format. You can continue statements over lines without continuation characters, statements can begin in any column, and there are no spacing rules.

Lexical Rules

SQL statements in a CA Ideal PDL program are delimited by EXEC SQL and END-EXEC. See the EXEC SQL statement.

The following are valid characters to use as delimiters for identifiers, reserved words, and numeric literals: space, comma, left parenthesis (, right parenthesis), less than <, greater than >, equals =, asterisk *, slash /, not Ø, ampersand &, vertical bar |, and colon :.

The apostrophe or quotation mark delimits alphanumeric literals. Double apostrophes or double quotation marks indicate a literal apostrophe or literal quotation mark.

Identifiers, literals, and reserved words can appear in the right-most and left-most columns of the source record.

Double hyphens are comment delimiters in embedded SQL. The colon is reserved for host-variable names in SQL. You cannot use it as a comment delimiter. See comments.

You cannot break words over the ends of lines.

You can leave lines blank.

For more information about permitted use, see the definitions of names, identifiers, literals, functions, and so on earlier in this chapter.

Comments

A comment is a character string that serves as documentation in a program. Comments are not executable. Any line in embedded SQL can contain a comment. The format of a comment is as follows:

-- text of comment

-- (double hyphens) Treats all characters to the right as a comment.

A line that begins with a double hyphen (--) is treated as a comment. Because the comment ends at the end of the line, no special character is required to terminate the comment.

Chapter 3: Procedure Definition Language Statements

This chapter describes the CA Ideal Procedure Definition Language statements in alphabetical order.

ADD Statement

The ADD statement increases the value of numeric fields. You can use ADD as an alternative to the SET statement.

This statement has the following format:

```
    {numeric field     }    {              }
ADD {numeric literal   } TO {numeric field }
    {alphanumeric field}    {date field    }
```

numeric field

Specifies a field with a type of N (numeric) or U (unsigned numeric) and a numeric value.
Limits: 31 digits

numeric literal

Specifies any series of 1 to 31 digits, with one optional decimal point and no embedded blanks, optionally preceded by a sign (+ or -).

alphanumeric field

Specifies a field with a type of X.
Limits: 32,000 characters

date field

Specifies a field with a type of D. The date field has a numeric value indicating an integer number of days from December 31, 1900 (day zero), plus or minus.

During execution both the source and the target fields must contain numeric values or a run-time error occurs.

ADD operands do not have to have the same decimal precision. When you add an expression with decimal places to a field with an integer value, the addition is performed and an attempt is made to put the result into the receiving field. If the value is too long, the decimal portion of the value is truncated. If the value that results from the truncation is still too long, a runtime error occurs.

You do not have to define the operands of an ADD statement with the same number of digits. However, an error occurs if the operation results in a value that has more significant digits than the second operand can contain.

Example

```
ADD MONTH_SALES TO YEAR_SALES
ADD 200 TO NET_INCOME
```

ASSIGN DATAVIEW Statement (CA Datacom/DB Native Access)

Use the ASSIGN DATAVIEW statement during execution of a single program to associate a CA Datacom/DB native access dataview with a database ID different from the DBID specified when the dataview was cataloged to CA Ideal, or to access a subset of a partitioned table. DBID or TABLE assignments established through this statement do not apply to other programs executed in the same run-unit. The ASSIGN DATAVIEW statement remains in effect for the program where it was issued throughout the run-unit until a subsequent ASSIGN DATAVIEW statement is executed in the same program for the same dataview.

This statement has the following format:

```
ASSIGN DATAVIEW name [ DBID dbid | TABLE tbl ]
```

name

Specifies the name of the dataview to associate with the database. You cannot use the abbreviation of DATAVIEW (DVW) in this statement.

dbid

A numeric literal or the identifier of a numeric or alphanumeric field that identifies the database with which the dataview is associated. The value must consist of three digits or three characters.

tbl

The three-character identifier of the child partition, or ANY set, for the partitioned table.

ASSIGN DATAVIEW remains in effect only for the duration of the program execution or until a different ASSIGN is made for the dataview. A dataview that a program uses is reassigned for a CA Ideal session if you issued an ASSIGN DATAVIEW command; however, the ASSIGN DATAVIEW statement overrides the ASSIGN DATAVIEW command.

You cannot issue an ASSIGN DATAVIEW statement in a FOR construct for that dataview.

An ASSIGN DATAVIEW statement applies only to the program that contains the statement, not to any calling or called programs associated with that program.

Example

```
LOOP VARYING I FROM 1 THRU 10
     ASSIGN DATAVIEW CLIENT DBID DBID_TABLE(I)
     FOR EACH CLIENT
         WHERE . . .
     .
     .
     .
     ENDFOR
ENDLOOP
```

ASSIGN REPORT Statement

This statement overrides the RUN command defaults or any current settings and permits report outputs to be handled individually.

This statement has the following format:

```
ASSIGN REPORT name [TO altname]

[                  {MAIL email id                } ]
[                  {LIBRARY                      } ]
[DESTINATION       { {SYSTEM 'name'   }          } ]
[                  { {NETWORK 'name' } [COPIES n] } ]

[DISPOSITION 'disp']

[MAXLINES m]

[DESCRIPTION 'string']

[DATE date_field]

[PAGE NUMBER page start]

[PAGE SIZE page-size]
```

name

The name of the report or the word RUNLIST for LIST statement output.

altname

An alphanumeric literal or the identifier of an alphanumeric field that specifies an alternate name for the report, as a ddname (in z/OS) or as SYSnnn (in VSE).

email-id

A 1- to 60-character alphanumeric literal or the identifier of an alphanumeric field that specifies the name of a [assign the value for emailp in your book] destination.

'name'

An alphanumeric literal (enclosed in single or double quotes) or the identifier of an alphanumeric field that specifies the name of a network or system printer destination.

Note: You cannot use a network printer destination in a batch run.

n

A numeric literal or the identifier of a numeric field that specifies the number of copies for a destination of system or network in an online environment only.

Note: The COPIES option is ignored in batch.

disp

An alphanumeric literal (enclosed in single or double quotes) or an alphanumeric field with the value of KEEP, HOLD, or RELEASE.

m

A numeric literal or the identifier of a numeric field that specifies the maximum number of lines of the report to produce. This value applies only to reports that are directed to the output library online.

'string'

An alphanumeric literal that describes the report.

date-field

A type D field that contains the date to use on the report where the report parameter date or the $RPT-DATE function is specified.

page-start

A numeric field or literal that contains starting page number of the report. This is the page number that is printed for report parameter page number or the $RPT-PAGE report function.

page-size

A numeric literal or the identifier of a numeric field representing the actual page size for the report. This value overrides the parameter specified in the report definition.
Limits: Maximum page-size is 250 lines per page, including heading and detail lines.

For more information about this statement available in command form, see the *Command Reference Guide*.

You can only issue ASSIGN REPORT when the report is not active, that is, before the first PRODUCE or after a RELEASE and before a subsequent PRODUCE.

BACKOUT Statement

The BACKOUT statement restores activity against tables and files accessed by the application to its most recent stable state. If no PDL CHECKPOINT, BACKOUT, TRANSMIT, SQL COMMIT, or ROLLBACK statement was previously executed, all updates in the run are removed.

BACKOUT applies to all CA Datacom/DB, DB2, and all recoverable VSAM files in CICS. (It does not apply to sequential files, panels, non-CICS VSAM files, panels, working data, or parameter data.)

This statement has the following format:

```
BACKOUT
```

Executing a BACKOUT statement executes the following:

In a CICS environment the BACKOUT statement executes a CICS SYNCPOINT ROLLBACK.

In a non-CICS environment:

- For VSAM files, the BACKOUT statement is ignored.
- For DB2 objects, the BACKOUT statement executes a SQL ROLLBACK.
- For native CA Datacom/DB or SQL access, the BACKOUT statement executes a ROLBK or LOGTB.
- If more than one database management system is accessed in the same run, the backouts against them are issued sequentially.

Execution of a BACKOUT statement has the same effect on a program as execution of an embedded SQL ROLLBACK and reverse. See the SQL documentation for the particular database for more information.

If a non-ideal subprogram executes an SQL ROLLBACK (batch) or a CICS SYNCPOINT (online), execute a CA Ideal BACKOUT or SQL ROLLBACK on return to the CA Ideal program.

For SQL Access in all Modes: After the execution of a BACKOUT statement in the logical scope of a FOR EACH or FOR FIRST construct, the set of rows is lost. Therefore, continued iteration of that FOR construct cannot resume. For more information, see the FOR EACH/FIRST Statement (SQL Access) topic in this chapter.

For VSAM: In a CICS environment, only files defined to CICS as recoverable are backed out.

The execution of a FOR EACH or FOR FIRST statement accessing a VSAM file can resume after a BACKOUT statement if the access uses at least one unique key. See the section titled FOR EACH/FIRST Statement (VSAM Files) in this chapter for further information.

PDL statements can continue to reference the columns processed by the last FOR statement after the BACKOUT.

Example

The following example sets a checkpoint after a FOR NEW construct adds records to the database and conditionally backs out the changes at the end of the procedure. Execution of the CHECKPOINT statement applies all modifications up to that point. A subsequent BACKOUT does not affect them. The BACKOUT only affects the FOR NEW insert. If you omit the CHECKPOINT, the BACKOUT rolls back all changes made since the TRANSMIT.

```
TRANSMIT
FOR EACH x
    statements : update of x
ENDFOR
FOR FIRST y
    statements : update of y
ENDFOR
FOR NEW z
    statements : add new z
ENDFOR
CHECKPOINT   :all modifications up to this point are applied
             :and not affected by a subsequent BACKOUT
FOR NEW w
    statements : add new w
ENDFOR
IF condition
    THEN BACKOUT
    ELSE CHECKPOINT
ENDIF
```

CALL Statement

A CA Ideal program can execute another CA Ideal program or a COBOL, PLI, or Assembler program. The program the first program executes is called a subprogram. A CALL statement passes control from a CA Ideal program to a subprogram and, optionally, passes data (in the form of input or update data items) between the two. After the called program terminates, control is returned to the calling program at the next sequential statement in the calling procedure.

Subprograms permit CA Ideal to access external routines and share procedures among several applications. The calling program references data items in the CALL statement. A data item can be the name of an elementary field, the name of a group, or a literal. The subprogram includes parameter definitions that describe these data items. CA Ideal manages the logical connections between the two.

For a full description of subprogram requirements and linkage conventions, see the *Creating Programs Guide*.

This statement has the following format:

```
CALL program_name [USING]  [INPUT data-item-1,…,data-item-n   ]…
                           [[UPDATE] data-item-1,…,data-item-n]
```

Program_name

Identifies the one- to eight-character user-defined name of the subprogram to invoke.

USING

An optional reserved word that you can add for readability.

INPUT

Indicates that the called program can reference, but not modify, the parameters that correspond to the data-items in the CALL statement. If you omit both reserved words INPUT and UPDATE, UPDATE is assumed until INPUT is specified for a subsequent parameter.

UPDATE

Indicates that the called program is allowed to modify the parameters that correspond to the data items in the CALL statement. If you omit the reserved word UPDATE, the default is UPDATE until INPUT is specified for a subsequent parameter.

data-item-1,...,data-item-n

Defines the data items to pass to the called program. Data items for which INPUT was specified can be literals or the identifiers of fields or groups (panels and dataviews are treated as groups). Data items for which UPDATE was specified must be identifiers of fields or groups (panels and dataviews are viewed as groups). You cannot use literals as UPDATE fields.

Examples

The following statement calls a program without passing data items.

```
CALL SUBPGM1
```

The following statement calls a program passing one UPDATE data item.

```
CALL SUBPGM2 A
```

The following statement calls a program passing one UPDATE data item.

```
CALL SUBPGM3 USING A
```

The following statement calls a program passing one UPDATE and two INPUT data items (both literals).

```
CALL SUBPGM4 A INPUT 'INIT', 23
```

The following statement calls a program passing one INPUT and one UPDATE data item.

```
CALL SUBPGM5 INPUT B UPDATE A
```

The following statement calls a program passing 5 UPDATE and 2 INPUT data items.

```
CALL SUBPGM6 A,B,C, INPUT D,E UPDATE F,G
```

For a detailed example that compares CA Ideal subprograms and non-ideal subprograms, see the *Creating Programs Guide*. For more examples of complete applications, review the sample applications and utility programs provided in source form on the installation tape. For more information about these sample applications, see the *Working in the Environment Guide*.

CHECKPOINT Statement

The CHECKPOINT statement commits all database activity, establishing the most recent stable state for tables and files the application accesses. You can subsequently recover this state with a BACKOUT statement. In long running batch jobs, perform periodic CHECKPOINTs.

A CHECKPOINT applies to all CA Datacom/DB tables, DB2 tables, and all recoverable VSAM files in CICS. It does not apply to sequential files, non-CICS VSAM files, panels, working data, or parameter data.

This statement has the following format:

```
CHECKPOINT
```

A TRANSMIT statement automatically causes a CHECKPOINT.

Executing a CHECKPOINT statement executes the following:

In a CICS environment the CHECKPOINT statement executes a CICS SYNCPOINT.

In a non-CICS environment:

- For VSAM files, the CHECKPOINT statement executes a TCLOSE.
- For SQL objects, the CHECKPOINT statement causes the execution of an SQL COMMIT.
- For CA Datacom/DB access, the CHECKPOINT statement causes the execution of a COMIT or LOGCP.
- If more than one database is accessed in the same run, the checkpoints against them are issued sequentially.

An SQL COMMIT statement is equivalent to a CHECKPOINT statement.

- If an SQL COMMIT is executed, ALL databases in the application will be CHECKPOINTed.

If a non-ideal subprogram executes an SQL COMMIT (batch) or a CICS SYNCPOINT (online), execute a CA Ideal CHECKPOINT or SQL COMMIT on return to the CA Ideal program.

Do not code a checkpoint statement in a FOR construct because the database update for each iteration of a FOR statement takes place at the ENDFOR. Placing the CHECKPOINT in the FOR construct causes a checkpoint before the database is updated, a subsequent BACKOUT would include only that record.

The CHECKPOINT statement executes a TCLOSE operation that flushes certain VSAM buffers if there was any VSAM access before the CHECKPOINT.

The execution of a FOR EACH or FOR FIRST statement accessing a VSAM file can resume after a CHECKPOINT statement only if the access uses at least one unique key. For more information, see the FOR EACH/FIRST Statement (VSAM Access) topic in this chapter.

For CA Datacom SQL ANSI Mode and DB2: After the execution of a CHECKPOINT statement in the logical scope of a FOR EACH or FOR FIRST construct, the set of rows is lost. Therefore, continued iteration of that FOR construct cannot resume. For more information, see FOR EACH/FIRST Statement (SQL Access) in the FOR Statement (SQL Access) topic in this chapter.

For VSAM: In a CICS environment, only files defined to CICS as recoverable will be affected.

In a non-CICS environment, the CHECKPOINT statement causes the execution of a TCLOSE operation, which flushes certain VSAM buffers if there was any VSAM access prior to the CHECKPOINT.

The execution of a FOR EACH or FOR FIRST statement accessing a VSAM file can be resumed after a CHECKPOINT statement only if the access uses at least one unique key. For more information, see the FOR EACH/FIRST Statement (VSAM Access) topic in this chapter.

Example

The following example commits the database after a FOR NEW construct adds records using dataview z.

```
FOR EACH x
   statements : update of x
ENDFOR
FOR FIRST y
   statements : update of y
ENDFOR
FOR NEW z
   statements : add new z
ENDFOR
CHECKPOINT
FOR NEW w
   statements : add new w
ENDFOR
```

Comment

A comment is a character string that serves as documentation in a program. Comments are not executable. Any line in a PDL program can contain a comment. If a line begins with : or --, the entire line is treated as a comment.

This statement has the following format:

```
:    text of PDL comment   outside EXEC SQL construct
--   text of comment       inside EXEC SQL construct
```

: (colon)-Treats all characters to the right as a comment. You cannot use the colon for comments in embedded SQL.

-- (double hyphens)-Treats all characters to the right as a comment.

Because the comment ends at the end of the line, no special character is required to terminate the comment. In embedded SQL, the colon is reserved for host-variable names. You cannot use it as a comment delimiter.

Example

```
: this is regarded as a comment.
SET A = B + C  :  this is regarded as a comment,
SET D = A + 1  -- and this, too, but not the SETs
```

DELETE Statement

The DELETE statement deletes the entire current record or row that an updateable dataview references. This applies only to SQL and CA Datacom/DB native access dataviews. You can use the DELETE statement only in a FOR FIRST, a FOR EACH, or a FOR ANY construct.

This statement has the following format:

```
DELETE dataview_name
```

dataview_name

Identifies the dataview where the current record or row is deleted.

DELETE does not apply to sequential or VSAM EDS files.

A DELETE is immediate and is not canceled by a QUIT in a FOR construct that contains the DELETE.

The entire record or row is deleted even if the dataview references only a subset of the fields in the record or row.

After a DELETE, you cannot reference the deleted record or row.

Example

```
FOR FIRST INVEN
    WHERE ITEM_NO = DESIRED_ITEM_NO
        DELETE INVEN
WHEN NONE
        DO INVALID_DEL
ENDFOR
```

DO Statement

The DO statement invokes another named procedure in the same program. Control is transferred to the named procedure and, when this procedure is completed, execution resumes with the statement that follows the DO statement in the invoking procedure.

This statement has the following format:

```
DO  {ERROR           }
    {procedure_name  }
```

ERROR

Invokes the error procedure and makes the $ERROR functions available. You can code this statement anywhere in the program procedure. You are responsible for resolving the error with a PROCESS NEXT or QUIT RUN statement.

Note: For more information about restrictions that apply to the error procedure, see the Error Procedure topic in this chapter.

procedure_name

The 1- to 15-character name of the invoked procedure.

Example

This example illustrates how DO statements invoke named procedures from another procedure. Each of the named procedures, ADD_REC, DEL_REC, and OTHER_PROC, is invoked when the select condition that precedes it is true.

```
<<MAIN>> PROCEDURE
      LOOP
         TRANSMIT MAINPNL
      UNTIL TRANSCODE = 'T'
         SET MAINPNL.MSG = ' '
         SELECT TRANS-CODE
         WHEN 'A'
           DO ADD_REC
         WHEN 'B'
           DO DEL_REC
         WHEN OTHER
           DO OTHER_PROC
         ENDSEL
      ENDLOOP
ENDPROC

<<ADD-REC>> PROCEDURE
      TRANSMIT ADDPNL CLEAR
      FOR NEW EMPLOYEE
        SET EMPLOYEE = ADDPNL BY NAME
        SET MAINPNL.MSG = 'EMPLOYEE ADDED'
WHEN DUPLICATE
        SET MAINPNL.MSG = 'RECORD ALREADY ON FILE'
      ENDFOR
ENDPROC
```

EJECT Statement

The EJECT statement causes the compilation listing to skip to the top of a page. The EJECT statement must be on a line by itself. It does not appear in the compilation listing.

This statement has the following format:

```
EJECT
```

Error Procedure

The error procedure specifies a set of actions to invoke whenever an execution-time error occurs. The error procedure also enables the program to provide for cases when certain conditions occur that could be handled and allow the program run to continue to completion.

For example, the error procedure can process an invalid numeric value condition and then return control to the processing procedure:

```
<<ERROR>> PROC
     IF $ERROR-CLASS = 'NUM'
        LIST ERROR
         PROCESS NEXT GET-NEXT-LOOP
     ELSE
        LIST ERROR
        BACKOUT
        QUIT RUN
     ENDIF
ENDPROC
```

Coding an error procedure is optional. It lets you process abnormal errors explicitly by overriding the default error procedure. The default error procedure does the following:

1. Issues a LIST ERROR statement that varies depending on the type of error.
2. Performs a BACKOUT.
3. Issues a standard message.
4. Performs a QUIT to end the run.

This statement has the following format:

```
<<ERROR>> PROCEDURE
     statements

 {ENDPROC      }
 {ENDPROCEDURE }
```

<<ERROR>>

A reserved label. Most statements that refer to labels cannot reference it; however, a DO statement can reference it. For example, QUIT ERROR is an illegal statement, but DO ERROR is valid.

statements

The action that takes effect when an error occurs. The statements in the body of the error procedure can consist of any PDL statements needed to process the error. The last statement is commonly a QUIT or PROCESS NEXT statement. If neither of these statements is issued, the default error procedure runs after the coded error procedure.

A program can contain only one error procedure. It can be anywhere in the program, but if it is the first procedure, the next procedure becomes the main procedure. When a procedure other than the first procedure becomes the main procedure, you must name the main procedure.

You can code $ERROR functions in the error procedure to return information about the last error. These functions only return meaningful data in the error procedure or in a procedure or CA Ideal subprogram invoked by the error procedure. The $ERROR functions are available directly in a CA Ideal subprogram called from the error procedure-you do not have to pass them as parameters. For more information regarding error handling, see the chapter “Error Handling“ in the *Creating Programs Guide*.

If you code an error procedure, you must code a BACKOUT statement to reverse the updates and inserts.

A QUIT statement without a label and a QUIT PROCEDURE statement are invalid in the error procedure because they imply a QUIT ERROR.

When an error with an $ERROR-CLASS of SYS occurs, your error procedure does not receive control and the default error procedure is invoked.

A runtime error in the error procedure invokes the default error procedure.

Generally, the default error procedure sets the value of the $RETURN-CODE function to 12 (unless it is already 12 or greater). The default error procedure runs after a coded error procedure that does not execute a QUIT RUN or PROCESS NEXT.

Notes for SQL Access

The error procedure is not invoked if the database management system detects errors in embedded SQL statements. They are governed by the WHENEVER statement in SQL. However, you can include a DO ERROR statement in embedded SQL to invoke the error procedure.

You can call the program @I$TIAR from the error procedure to format SQL codes into text messages. For more information regarding non-Ideal utility programs, see the Utility Programs appendix.

Example

Following is CA Ideal default error processing:

```
<<ERROR>> PROCEDURE
       IF $RC LT 12
          SET $RC EQ 12
       ENDIF
       LIST ERROR
       BACKOUT
       QUIT RUN
       ENDPROC
```

This is an example of an error procedure that you can use in your CA Ideal program:

```
<<ERROR>> PROCEDURE
        DO LOG-MSG
        PROCESS NEXT EMP-DVW
ENDPROC
```

EXEC SQL Statement (SQL Access)

EXEC SQL delimits an embedded SQL statement. The end of the SQL statement is marked by END-EXEC.

This statement has the following format:

```
EXEC SQL
       SQL-statement
END-EXEC
```

You can code the EXEC SQL statement, the SQL statement, and the END-EXEC on a single line or on multiple lines. You can omit the END-EXEC if the SQL statement is immediately followed by another EXEC SQL statement. For more information about embedded SQL, see the "Procedure Definition Language Statements" chapter.

Example

```
EXEC SQL
      UPDATE DELINQUENT_ACCT
        SET ACCT_NO = :PNL_ACCT_NO
        WHERE PAST_DUE GT 90
END-EXEC
```

FOR Constructs (CA Datacom/DB Native Access)

The FOR statement is used for reading and updating the database. The FOR construct begins with a FOR statement and ends with an ENDFOR statement.

To process data from the database, you must first define a CA Ideal dataview for the data. The dataview defines the fields that are available to the application. For a description of CA Ideal dataviews, see the *Creating Dataviews Guide*.

Set Processing

The FOR EACH, FOR FIRST, and FOR ANY constructs retrieve and update a set of records. These constructs are iterative. With each iteration, it returns the next record in the requested set. It is not necessary to create an image of the record in working data since the dataview referenced in the FOR construct contains a data structure to hold the record retrieved by each iteration of the FOR. PDL statements can use the fields in this structure. The database is automatically updated at the ENDFOR for the current iteration.

Inserting Records

Use the FOR NEW statement to insert a new record into the database. The FOR NEW statement adds a single record and is not iterative. To repeat processing of a FOR NEW statement, include it in a looping construct.

Exclusive Control

Exclusive control is an enqueuing mechanism CA Datacom/DB provides to protect database records against the following:

- Destructive simultaneous update by two more different tasks.
- This is governed by primary exclusive control.
- Two tasks consecutively update the same record, followed by an abend and transaction backout by one of those tasks. Without exclusive control, the update by the unaffected task might be backed out without the task being aware of it.
- This control is provided by secondary exclusive control.

Exclusive control operates at the logical record level. If Task 1 attempts to read a record for update that is held under either primary or secondary exclusive control by Task 2, it waits until Task 2 releases control of the record. Task 1 cannot get update control of the record until one of the following occurs:

- Task 2 completes.
- Task 2 issues a CHECKPOINT, TRANSMIT or CICS SYNCPOINT command.
- Task 2 abends and transaction backout completes.

If Task 2 is a long running batch job that is updating many database records without issuing CHECKPOINT commands, all online tasks that access those same records for update wait. This can degrade online response time.

Primary Exclusive Control

The dataview in the following example is an updateable dataview. (In all the examples that follow, 'FOR EACH DVW' can be a full FOR statement including WHERE and ORDERED BY clauses).

Example

```
FOR EACH DVW
        SET...
        SET...
ENDFOR
```

Each record is read and returned to the program one by one. Each record is read with primary exclusive control. At this point, the record is updated. While it is held under primary exclusive control, another task cannot simultaneously read the record with update intent.

Primary exclusive control is established when the record is selected as part of the set of requested records. Primary exclusive control is maintained until after the last statement in the construct before the ENDFOR. The important points to note in the above example are:

- The FOR statement is processing only one record of the set at any point in time.
- Only one record of the set that meets the FOR/WHERE/ORDERED BY criteria is under primary exclusive control at any point in time.
- Primary exclusive control begins when the record is read at the FOR statement and ends when the update is done just before the ENDFOR statement.
- Primary exclusive control also ends if a TRANSMIT or a CICS SYNCPOINT command is encountered.
- A record held under primary exclusive control by one task cannot by simultaneously read with update intent by another task.

Note: This description does not apply to sequential batch processing.

Secondary Exclusive Control

Secondary exclusive control gives the application control over when the update of a group of dependent records is committed.

For instance, you might need to update a master record with totals from three different detail records. The processing requires that the three detail records and the master record either all be updated or all backed out. Data integrity is lost if two of the detail records were updated and an error was discovered on the third detail record. In this case, the update of the first two detail records should be backed out, an appropriate error message issued to the terminal operator, and the master record update never attempted.

When the record is updated just before the ENDFOR, primary exclusive control ends and secondary exclusive control begins. Thus, as each record is processed and updated, it is added to the group of records the task holds under secondary exclusive control.

In the above example, if the URT specifies TXNUNDO=YES and the MUF master list specifies LOGGING=YES, secondary exclusive control is in effect, in addition to primary exclusive control.

The important points to note about secondary exclusive control in the example are:

- A task can have many records under secondary exclusive control at one time.
- As the records that meet the FOR/WHERE/ORDERED BY criteria in the set are processed and updated, they are added to the group of records held under secondary exclusive control. When the first record of the set is read with primary exclusive control, no records of the set are held under secondary exclusive control. When the last record of the set is updated, all the records of the set are held under secondary exclusive control. See the next point for an exception.
- Secondary exclusive lasts until the task terminates, until a TRANSMIT is issued, until a CHECKPOINT is issued, or until a CICS SYNCPOINT is issued.
- While a record is held under secondary exclusive control, no other task can read that record with update intent. Therefore, another task cannot simultaneously delete or update the record.
- Generally, if a record is under primary exclusive control, it is not under secondary exclusive control. However, it is possible for a task to hold the same record under both primary and secondary exclusive control.

This can occur if the task reads the record with update intent (through the CA Datacom/DB commands RDUKY, SELFR, or SELNR), updates the record (UPDAT), and reads the record again with update intent (RDUKY, RDUID, SELSM, SELFR, and so on). This occurs in CA Ideal in the following situation:

```
FOR DVW-UPDATE
          WHERE num-value = 'nnnnn'
          MOVE ...   to DVW-update-field
          MOVE ...   to DVW-update-field
ENDFOR
FOR DVW-UPDATE
      WHERE num-value = 'nnnnn'
          ...
ENDFOR
```

If the value for *num-value* were the same for both FOR statements, the record is read for update and updated by the first FOR construct. It is then read again (using the SELFR DB command) for update. At this point, it is held under both primary and secondary control.

```
FOR EACH DVW1
        FOR EACH DVW2
                SET ...
                SET ...
        ENDFOR
        SET ...
        SET ...
ENDFOR
```

In this example, the FOR statement for dataview DVW2 is nested in the boundaries of the FOR statement for dataview DVW1. Once the FOR statement for DVW2 is encountered, two records are held under primary exclusive control at the same time, one record from DVW1 and one record from DVW2. As the two sets of records are processed, records from both sets are added to the records held under secondary exclusive control by this task.

```
FOR EACH DVW
        CHECKPOINT
        SET ...
        SET ...
ENDFOR
```

In this example, a CHECKPOINT is issued in the boundary of the FOR/ENDFOR construct. A CHECKPOINT applies to all records held under secondary exclusive control.

This means that, because each record is held under primary exclusive control while it is being read, the CHECKPOINT does not commit the record while it is read. Not until the next iteration of the FOR, when the next record is read, does the previous record come under secondary exclusive control and become committed.

The important points to note here are:

- When the CHECKPOINT statement is encountered, all updated records currently under secondary exclusive control are committed. That means that if the task abends, these updates are not backed out.
- As a result of the CHECKPOINT command, other tasks can update all records committed by the CHECKPOINT statement.
- If there are multiple FOR statements preceding the CHECKPOINT command, the CHECKPOINT command releases all records held under secondary exclusive control by those FOR statements.

```
FOR EACH DVW1
         CHECKPOINT
         FOR EACH DVW2
                  SET ...
                  SET ...
         ENDFOR
         SET ...
         SET ...
ENDFOR
```

This FOR construct illustrates the example of the master record (dataview DVW1) and its corresponding detail records (dataview DBW2) committed if all are successfully updated. If the update of the detail records proceeds normally and the master record is also updated successfully, a CHECKPOINT command is issued to commit the successful updating of the dependent records. The master and detail records are released from secondary exclusive control making them available for updating by other tasks.

```
FOR EACH DVW
         SET ...
         SET ...
         TRANSMIT panel
         SET ...
         SET ...
ENDFOR
```

In this example, a TRANSMIT statement is issued in the scope of the FOR construct. The TRANSMIT releases both primary and secondary exclusive control. If a record is held under secondary exclusive control, it is checkpointed. If a record is held under primary exclusive control, it is released.

When the run continues normally, the update logic detects at the ENDFOR that the record was released due to a TRANSMIT, rereads the record to re-establish primary exclusive control, and updates it. If another user task read and updated the record between the TRANSMIT and the update of the record, a CA Ideal run-time error with a $ERROR-DVW-STATUS of I3 is issued. If another user task deleted the record between the TRANSMIT and the update of the record, an CA Ideal run-time error with a $ERROR-DVW-STATUS of I2 is issued.

If the task abends after the TRANSMIT, the records that were updated and held under secondary exclusive control are not backed out because the TRANSMIT checkpoints them.

The important points to note here are:

- A TRANSMIT releases all records currently held under either primary or secondary exclusive control by the task.
- If the TRANSMIT is in the scope of a FOR construct, CA Ideal ensures the integrity of data for records read for update across transaction boundaries (that is, those that were held with primary exclusive control at the time of the TRANSMIT).

Batch Processing

Two considerations for Batch processing are as follows:

- Restarting Programs
- Sequential Processing

Restarting Programs

An update to a CA Datacom/DB record places it under secondary exclusive control until a LOGCP request is issued. For online users to wait for updates as little as possible, batch CA Ideal programs should release control by issuing CHECKPOINT statements as often as possible.

This means that you must write batch programs doing database updates so that they are restartable at each CHECKPOINT.

One of the easiest ways to do this is to first load the transaction records onto a CA Datacom/DB table. As each transaction record is successfully processed, a record confirming the successful update is written to another table, the transaction record is deleted, and a CHECKPOINT taken. Any transactions that fail are updated to show this or are transferred to a reject table.

If there is a system failure, you can simply rerun the programs using whatever transaction records remain in the table. These are all of the unprocessed transactions. You can back out any confirmation data from an incomplete update. Finally, you can use the confirmation data to produce a report of the run. Once the report is complete, the table can be cleared.

This method produces the report of the run from a confirmation table after doing the updates. This lets you restart the update without having to restart the report (a report is harder to restart because totals must be resumed, control breaks re-established, and so on). It also lets you rerun the report, for example, if it is lost after production, without re-updating the database.

Sequential Processing

CA Ideal can do multi-block read-aheads of a CA Datacom/DB non-SQL table using the GETIT command if certain conditions are met. For more information, see the *CA Datacom/DB* Database *and System Administrator Guide*. In this case, exclusive control is acquired for an entire block of records as it is read, even though the program might not be updating all records. If you also use AUTODXC=NO, the exclusive control is maintained up to the next CHECKPOINT, if any.

There is no way to request a release of exclusive control for a record in that block, even if the application did not read it. So do not use sequential processing if your program needs to have a QUIT in any FOR construct because you cannot release exclusive control of records in the block that were not yet accessed by the FOR.

Even read-only applications can acquire exclusive control for a block of records. This is because such applications need to run with UPDATE=YES so that Compound Boolean Selection access can update the database index for the table. Again, do not use sequential processing if your program needs to have a QUIT in any FOR construct.

FOR EACH/FIRST/ANY Statement (CA Datacom/DB Native Access)

The FOR EACH, FOR FIRST, and FOR ANY statements process a set of records (or process a single record) from a CA Datacom/DB table. All of the statements in the FOR construct apply to each record selected.

The FOR construct is iterative. With each iteration, it returns the next record in the requested set. FOR FIRST, FOR EACH, and FOR ANY are the only constructs that update or delete a record.

To process data from the database, you must first define a dataview to CA Ideal for the data. The dataview defines the fields that are available to the application.

This statement has the following format:

```
[<<label>>]
    [EACH            ]
FOR [ALL             ] dataview_name[NO UPDATE]
    [[THE] FIRST [n]  ]
    [ANY n           ]

[WHERE where condition]

[                    [ASCENDING ]             ]
[ORDERED BY [UNIQUE] [DESCENDING] id [[,]id]… ]
[        [[ASCENDING ]                        ]
[        [[DESCENDING] id [[,] id…]           ]

      statements
[WHEN NONE    ]
[   statements]

[WHEN ERROR    ]
[    statements]

ENDFOR
```

<<label>> (Optional)

Specifies 1- to 15-character name of the FOR construct. You can refer to the construct in QUIT and PROCESS NEXT statements and as the operand of certain functions such as $COUNT.

EACH|ALL

Indicate that the statements in the FOR construct apply to every record that satisfies the where condition. You can use the reserved words EACH and ALL interchangeably.

[THE] FIRST [n] (Default)

Specifies that the statements in the scope of the FOR construct apply to the first n records that satisfy the where condition. The value specified for n can be an identifier of a numeric field or a numeric literal that specifies the number of records to process. The default is FIRST 1. You can add the reserved word, THE, for readability.

When you use FOR FIRST n with a where condition and an ORDERED BY clause, all records that satisfy the where condition are ordered and then the first *n* ordered records are selected. The difference between FOR FIRST and FOR ANY is illustrated in the examples in this section.

ANY n

Specifies that the statements in the scope of the FOR construct apply to any n records that satisfy the where condition. The value specified for n can be an identifier of a numeric field or a numeric literal that specifies the number of records to process. The value of n is required for FOR ANY.

When you use FOR ANY n with a WHERE condition and an ORDERED BY clause, the first n records that satisfy the WHERE condition are selected and then ordered. The difference between FOR FIRST and FOR ANY is illustrated in the examples in this section.

dataview_name

The name of the dataview processed.

NO UPDATE (Optional)

Specifies that the records processed by this FOR construct are not updated and, therefore, are not held under exclusive control. This applies even if the dataview is defined as updatable in CA Datadictionary. If used, this clause must immediately follow the dataview name. You can use the SET RUN UPDATE command to temporarily suppress updates; however, SET RUN UPDATE is primarily intended for testing purposes. For more information, see the *Command Reference Guide*.

WHERE clause(Optional)

Specifies that the statements in the scope of the FOR construct apply to those records that satisfy the specified condition.

where-condition

A condition (as defined in the PDL Language Elements section in chapter SQL Concepts and Language Elements with the following further qualifications:

- The left-hand operand of each relational-expression must be the identifier of a field or group in the dataview being referenced.
- If the left-hand operand is an alphanumeric field, the right-hand operand must be an alphanumeric expression.
- If the left-hand operand is a numeric field, the right-hand operand can be a numeric expression, an alphanumeric expression, or a non-alpha group that is not a panel group or dynamic matching parameter. When the right-hand operand is not a numeric expression, a warning is issued when the program is compiled and, if the right-hand operand cannot be converted to numeric, a runtime error occurs.
- A field name used as the left-hand operand of a relational-expression in a where condition does not need to be qualified with a dataview name since it refers implicitly to the dataview in the FOR clause. However, reserved words used as operands must always be qualified.

The right-hand operand of a relational-expression can be any arithmetic or alphanumeric expression, but cannot reference any fields in the dataview named in the FOR clause.

A where-condition is the only condition that can contain the special relational operators CONTAINS and NOT CONTAINS. For an explanation and examples of CONTAINS and NOT CONTAINS, see the definition of the where-condition in the PDL Language Elements section in "SQL Concepts and Language Elements" chapter and the $FIXED-MASK function in "Symbolic Debugger Commands" chapter.

If the condition is a condition name, it must be from the dataview being referenced. If the condition name is used for more than one data structure, the condition name must be qualified.

Where-conditions cannot be Boolean functions or flags.

Any subscripts used in the where-condition must not be numeric fields in the dataview being referenced.

ORDERED BY clause (Optional)

Determines the logical order in which the records are processed. If this clause is omitted, the dataview records are processed in an optimal order.

Note: This optimal order is determined dynamically at program execution time and can change based on the contents of the database and by the release level of the DBMS.

UNIQUE (Optional)

Specifies that only one record with each unique value of the ORDERED BY identifiers is processed.

ASCENDING/DESCENDING id (Optional)

Specifies whether the identified fields are processed from low value to high value (ASCENDING) or high value to low value (DESCENDING). ASCENDING is the default and applies to each identified field until DESCENDING is specified. DESCENDING then remains in effect for each additional identified field until ASCENDING is specified again.

The effect of ASCENDING/DESCENDING depends on the type of the identified field. Type X fields are ordered in ascending or descending EBCDIC order. Type N and type D fields in ascending or descending numeric order.

id

The identifier of a numeric or alphanumeric field or alpha-group. Identifiers can be subscripted, but not by fields in the dataview being referenced.

Statements

PDL statements. The statements in the logical scope of a FOR construct can reference any field in the record most recently processed by the FOR.

WHEN NONE

An optional postscript that specifies that when none of the records meets the where condition, the statements following the WHEN NONE are executed.

WHEN ERROR (Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, user-defined or default error procedures process the errors.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions no longer are available. For more information about WHEN ERROR processing, see the following examples in this section.

Note: Only the WHEN ERROR clauses handles dataview errors ($ERROR-CLASS=DVW). User-specified or default error procedures handle system and internal errors.

ENDFOR

A reserved word that marks the end of the FOR construct. If FOR statements are nested, the most recent undelimited FOR construct is delimited by the first occurrence of ENDFOR. Each FOR in a nested FOR construct must have a corresponding ENDFOR.

The actual update takes place at the ENDFOR for the current iteration for all changes except deletes.

Fields processed by each iteration of the FOR construct can be referenced in PDL statements. The identifier is the name of the field defined in the dataview or the field name with the dataview name as qualifier. For example, field ACCT_NO in dataview ACCT can be compared to the field ACCT_NO in panel PNL1:

```
IF ACCT.ACCT_NO EQ PNL1.ACCT_NO ...
```

You cannot make such references before the FOR is executed.

Sometimes it is convenient to first process a dataview record and then refer to its fields, rather than coding the actions in the FOR. For example, you can delegate finding the appropriate record to a lower level procedure.

Statements outside the logical scope of the FOR construct can access but not update data in the dataview record. The values of the fields processed by the most recent iteration of the FOR are still available after the ENDFOR (see examples at the end of this section), except when the record was deleted or no records were found (WHEN NONE).

Data in the dataview record is available until another FOR accesses the same dataview record.

Updates (changes and deletes) must be done in the logical scope of the FOR. Any update of a dataview field in the logical scope of a FOR virtually updates the database. For changes (but not for deletes), the actual update takes place at the ENDFOR for the current iteration. Therefore, any QUIT or PROCESS NEXT executed in the scope of a FOR abandons the update of the current record even for fields whose values already were changed with SET or MOVE statements and a checkpoint in the logical scope of the FOR does not commit the current update because the update does not take place until the ENDFOR.

If statements outside the logical scope of the FOR construct attempt to update the record (with a SET, MOVE, and so on), an execution-time error results.

CA Ideal suppresses database writes when it can determine that the data was not altered. If a call is made to a non-ideal subprogram that updates the database, specify UPDATES DB or UPDATES DB2 on the subprogram identification panel (described in the *Creating Programs Guide*). This insures that CA Ideal does not suppress CHECKPOINT, BACKOUT, ROLLBACK, or SQL COMMIT statements in any transaction where the subprogram is called. If statements outside the logical scope of the FOR construct attempt to update the record (with a SET, MOVE, and so on), an execution-time error results.

Changing the value of a field in the logical scope of a FOR construct has no impact on the selection of the next record since selection is made at the time the FOR block is initially entered.

You can nest any of the FOR constructs as long as each FOR construct refers to a different dataview. You cannot nest a FOR construct for a given dataview in another FOR construct for the same dataview.

If the dataview was defined as updateable, the logical scope of the FOR construct is implicitly the scope over which each successive record is held exclusively. A transmit in the FOR construct releases exclusive control. After the transmit, CA Ideal ensures the integrity of the record by rereading it and causing an error if it was changed. A transmit in the FOR construct releases exclusive control; after the transmit, CA Ideal/PC ensures the integrity of the record by rereading it and causing an error if it was changed.

When a QUIT is executed in the logical scope of a FOR, the next statement executed is the statement after the ENDFOR. When FOR and LOOP constructs are nested, any construct can be abandoned by referencing the optional label in a QUIT statement. (See the QUIT statement in this chapter).

If more than one position (record) of a dataview is needed simultaneously, do one of the following:

- Use two dataviews for the same file.
- Save necessary information in working data.

You can use the WHERE and ORDERED BY clauses in either order.

You can improve efficiency by using fields that are keys in WHERE clauses. ORDERED BY clauses are most efficient when the sequence of the fields in the clause matches the sequence of the fields in a complete key.

Do not nest a FOR construct for a given dataview in another FOR construct for the same dataview. However, you can nest a FOR NEW for the same dataview in the WHEN NONE of the outer FOR.

Example

```
FOR EACH DELINQUENT-ACCT
    WHERE BALANCE > 200
       DO CONTACT-COLLECTOR  :if qualification needed,
ENDFOR                       : use DELINQUENT-ACCT.field
```

Example

```
<<EMP-SEARCH>>
FOR EACH EMPLOYEE
    WHERE DEPT='D' AND JOB-CODE = 'J'
       DO CHECK-GOOD-EMP
        IF ENOUGH-GOOD-EMP
             QUIT EMP-SEARCH
       ENDIF
ENDFOR
```

Example

```
FOR FIRST INVENTORY-ITEM
    WHERE QOH > 50 AND PRICE < 500
        DO PROCESS-ITEM
ENDFOR
```

Example

```
FOR THE FIRST 5 INVEN
    WHERE PRICE < 100
        DO P-5-CHEAP-ITEMS
ENDFOR
```

Example

```
FOR EACH EMPLOYEE
    WHERE DEPT = 'D'
       FOR EACH PAY-REC
            WHERE PAY-REC.EMP-NO = EMPLOYEE.EMP-NO
                DO PROCESS-PAY
        ENDFOR
ENDFOR
```

Example

```
FOR FIRST ACCT
    WHERE PAST-DUE > 90
    ORDERED BY ACCT-NO
    : you can refer to or update "ACCT.field" here
      WHEN NONE
    DO NO-DELINQ-ACCT
       ENDFOR
        : you can now refer to "ACCT.field" if present
        : you cannot update "ACCT.field" here (unless this
        : is a procedure performed by a DO from in the
        : FOR)

DO FIND-CUSTOMER
       IF CUST-FOUND
     : you can refer to "CUST.field" here
       ELSE
    DO CUST-NOT-FOUND
ENDIF
```

Example

```
<<FIND-CUSTOMER>> PROCEDURE
FOR THE CUST
    WHERE CUST-NO = TRANS-CUST-NO
        SET CUST-FOUND = TRUE
WHEN NONE
        SET CUST-FOUND = FALSE
ENDFOR
```

Example

```
FOR FIRST 20 ITEMS
    WHERE UNIT-PRICE > 10
    ORDERED BY SHORT-DESC
         LIST ITM-ID, SHORT-DESC, UNIT-PRICE
 WHEN NONE
    DO INCREASE-PRICE
ENDFOR
```

Returns...

```
A60009  ADAPTER    24.99
A70002  ANTENNA    19.99
A70003  ANTENNA    19.99
O10002  ARMCHAIR  304.00
H20000  BEDBOARD   54.99
H20002  BEDWEDGE   18.99
```

Example

```
FOR ANY 20 TIMES
    WHERE UNIT_PRICE > 10
    ORDERED BY SHORT_DESC
       LIST ITEM_ID, SHORT_DESC, UNIT_PRICE
WHEN NONE
    DO INCREASE_PRICE
ENDFOR
```

Returns...

```
A30001  CADDY     89.99
A30000  CONSOLE   39.99
A40001  COVER     19.99
A40002  COVER    199.99
A40003  COVER    199.99
A50001  CUSHION   14.99
```

Example

```
FOR FIRST CUSTOMER
  WHERE CUSTID = PNL-CUST
    DELETE CUSTOMER
WHEN NONE
  NOTIFY 'NO CUSTOMERS FOUND'
WHEN ERROR
  SELECT FIRST ACTION
    WHEN $ERROR-DVW-STATUS = 94 AND
         $ERROR-INTERNAL DVW-STATUS = 31
      LIST 'Constraint Error: ' $ERROR-CONSTRAINT-NAME
      NOTIFY 'Customer ' CUSTID 'has open orders and cannot be deleted'
   WHEN $ERROR-DVW-STATUS = 36
        NOTIFY 'Contact Database Administrator with error information'
   WHEN OTHER
        DO ERROR
  ENDSEL
ENDFOR
```

FOR NEW Statement (CA Datacom/DB Native Access)

The FOR NEW statement can insert a new record into a CA Datacom/DB table. This statement uses a native command dataview defined to access the table. The FOR NEW statement is not iterative; to repeat processing of a FOR NEW, you must include it in a looping construct.

This statement has the following format:

```
<<label>>
     FOR [THE] NEW dataview_name
            statements

   [WHEN DUPLICATE ]
   [    statements ]

   [WHEN ERROR     ]
   [    statements ]
  ENDFOR
```

[<<label>>]

An optional 1- to 15-character name of the FOR NEW construct. You can use it to refer to the construct in a QUIT statement.

FOR [THE] NEW

Specifies the action to take to insert or add each new record.

FOR NEW initializes the field values in the new record if the program does not initialize them. The column values are initialized to:

- NULL for fields that can have the null value
- Zeros for numeric fields
- Zero length for variable-length fields
- Spaces for alphanumeric fields
- Current time for time fields
- Current date for date fields
- Current timestamp for timestamp fields

If initial values were specified for the field in the dictionary, they are used.

Note: CA Datacom/DB initializes fields to spaces in the underlying record that are not defined in the dataview without regard to the intended data type of the field. Therefore, dataviews used in FOR NEW should span the entire record.

You can add the reserved word, *THE,* for readability.

dataview-name

The name of the dataview that defines the new record inserted. The dataview must be updateable.

statements

PDL statements. Typically, the statements in the scope of the FOR NEW construct are those that place values into the newly created record.

WHEN DUPLICATE (Optional)

The WHEN DUPLICATE clause contains statements that are executed when the key value of a record to add matches the key value of a record existing in the database and when the database does not allow duplicate key field values.

If the WHEN DUPLICATE clause is omitted and duplication is not allowed, control passes to the WHEN ERROR statements when a duplicate record is found. If WHEN ERROR is not coded, control passes to the error procedure.

If the WHEN DUPLICATE clause is included and duplication is allowed, the WHEN DUPLICATE clause is ignored.

Note: Although the file is not updated when a duplicate record is found (the duplicate record is not added), the WHEN DUPLICATE clause does not affect the execution of the statements that precede it. The statements in the WHEN DUPLICATE clause are executed when the duplication is detected at the ENDFOR. At that point, all other statements in the scope of the FOR have already executed. If the FOR construct includes statements that increment counters or set messages, you can correct those values in the WHEN DUPLICATE processing. However, you cannot continue executing the FOR construct.

WHEN ERROR (Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors ($ERROR-CLASS=DVW) are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error processing.

ENDFOR

A reserved word that terminates the FOR construct. If FOR constructs are nested, the most recent unterminated FOR construct is terminated. You can reference any recently added field in the dataview record after ENDFOR unless a QUIT statement is used.

A QUIT in the logical scope of a FOR NEW abandons the creation of the record. Further reference to fields in the dataview outside of the FOR is invalid.

Insertion of a new record into the database occurs at the ENDFOR.

You cannot delete a record defined by the dataview specified in the FOR NEW construct in the logical scope of the FOR NEW construct.

If inserting the new row causes a CA Datacom/DB abnormal error, the WHEN ERROR statements executed. If a WHEN ERROR statement is not coded, the error procedure gets control at the ENDFOR.

You can nest FOR EACH in WHEN DUPLICATE.

If the record contains a SQL DATE, TIME, or TIMESTAMP field, the field is set to the system date, time, or timestamp before it is converted and written to the database.

Example

In the following example, the FOR NEW construct is included in a LOOP construct to process multiple records. Notice that a WHEN DUPLICATE clause is specified to correct the NEW_COUNT total when the duplicate record was not added.

```
LOOP UNTIL TRANSCODE = 'Q'
      TRANSMIT INVEN_PNL
      FOR THE NEW INVEN_ITEM
            MOVE INVEN_PNL TO INVEN_ITEM BY NAME
        SET NEW_COUNT = NEW_COUNT + 1
      WHEN DUPLICATE
            SET NEW_COUNT = NEW_COUNT - 1
      ENDFOR
        ENDLOOP
```

FOR Statement (SQL Access)

The FOR statement is used for reading and updating the database**.** It is an alternative to coding embedded SQL statements. The FOR construct begins with a FOR and ends with an ENDFOR.

To process data from the database using a FOR construct, you must first define a CA Ideal dataview that identifies an SQL object (table, view, or synonym).

The FOR EACH and FOR FIRST constructs retrieve and update rows from the table or view. These constructs are iterative. With each iteration, they return the next row in the requested set. If a row is updated in the logical scope of the FOR (and updating is allowed), the database is automatically updated at the ENDFOR for the current iteration. SQL UPDATE statements are not needed.

You can use the FOR NEW statement to insert a new row into the table.

A CA Ideal data structure is automatically generated for the row that each iteration of the FOR retrieves. The same data structure is used by any FOR accessing the same SQL object. The fields in this group are identified by the names of columns processed. PDL statements can use them and embedded SQL statements use them as host variables. Embedded SQL statements can also be used independently of a FOR construct to fetch data directly into host structures in working data, parameter data, or panels and to update the database.

For more information about establishing and maintaining dataviews for SQL access, see the *Creating Dataviews Guide*. For more information about describes preparing and maintaining application plans and packages for DB2 access or access plans for CA Datacom SQL access, see the *Administration Guide*. For a description about SQL syntax and language elements, see the "Procedure Definition Language Statements" chapter.

FOR EACH/FIRST Statement (SQL Access)

The FOR EACH/FIRST statement processes a set of rows (or process a single row) from an SQL object. All of the statements in the logical scope of the FOR construct apply to each row selected. The FOR construct is iterative. With each iteration, it returns the next row in the requested set.

To process data from the database, you must first define a CA Ideal dataview for the data. The dataview identifies the table, view, or synonym to CA Ideal.

This statement has the following format:

```
[<<label>>]
    [EACH            ]
FOR [ALL             ] dataview_name [NO UPDATE]
    [[THE] FIRST [n]]

[WHERE search-condition]

[           [ASCENDING  ]  column [[,] column]...  ]
[ORDERED BY [DESCENDING ]                          ]
[                                                  ]
[[[ASCENDING    ]                      ]           ]
[[[DESCENDING   ] column [[,] column...] ]         ]
           statements
[WHEN NONE     ]
[    statements]
[WHEN ERROR    ]
[    statements]

ENDFOR
```

<<label>>

An optional 1- to 15-character name of the FOR construct. You can use this label to refer to the construct in QUIT and PROCESS NEXT statements and as the operand of certain functions such as $COUNT.

EACH|ALL

Indicates that the statements in the scope of the FOR construct apply to every row that satisfies the search condition. The reserved words EACH and ALL can be used interchangeably.

[THE] FIRST [n] (Default)

Specifies that the statements in the scope of the FOR construct apply to the first *n* rows that satisfy the search condition. The value specified for *n* can be an identifier of a numeric field or a numeric literal that specifies the number of rows to process. The default is FOR FIRST 1. You can add the reserved word THE for readability.

When you use FOR FIRST *n* with an ORDERED BY clause, the rows that satisfy the search condition are ordered and then the first *n* ordered rows are selected.

FOR FIRST generates an OPTIMIZE FOR clause on the SQL SELECT statement. If you specify a literal for *n*, the literal is used in the OPTIMIZE FOR clause. If you specify a host variable for *n*, the number generated for the OPTIMIZE FOR clause depends on the defined size of the host variable:

- If the host variable is defined as a single digit, the OPTIMIZE is set for 9 rows.
- If the host variable is defined as a two-digit number, the OPTIMIZE is set for 99 rows, and so on.

dataview_name

The name of the dataview defined for the table, view, or CA Datacom/DB synonym processed.

Note: Do not qualify the dataview name with an authorization ID.

NO UPDATE (Optional)

Specifies that the rows processed by this FOR construct are not updated. If NO UPDATE is specified in the FOR construct, FOR FETCH ONLY is included in the generated SQL statements. If used, this clause must immediately follow the dataview name. See also the SET RUN UPDATE command.

WHERE clause(Optional)

Specifies that the statements in the scope of the FOR construct apply to those rows that satisfy the search condition.

search-condition

Specifies a condition that conforms to the SQL syntax for a search condition with the following qualifications:

You can use SQL functions in a search condition where SQL rules allow the functions. You cannot use PDL built-in functions in a search condition.

You can use the following predicates in a search condition: IN, BETWEEN, LIKE, NOT IN, NOT BETWEEN, NOT LIKE, IS NULL, and IS NOT NULL. You cannot use the CONTAINS predicate in a search condition.

The LIKE condition can include the ESCAPE option to change the mask character. This allows the default SQL mask characters % and _ to be present in the data being searched.

You can use the keyword CONCAT in place of the concatenation operator, ||.

You cannot include SQL subqueries in a search condition. For this reason, you cannot use the EXISTS predicate.

A search condition requires the CA Ideal syntax for alphanumeric literals and comments rather than the SQL syntax. Hexadecimal literals, for example X'FFFF' are not supported in search conditions.

Host variable names are specified without an initial colon in search conditions.

The search condition can include column names from the specified object and host variable names on either the left or right side of the predicate.

Column names can only be qualified by table or view names to two levels *(table_name.col_name*) in the search condition.

You cannot use implied predicate subjects and operators or subscripted identifiers in a search condition. You cannot use numeric dynamic match parameters in a search condition unless a default precision was specified.

ORDERED BY clause (Optional)

Determines the logical order in which the rows are processed. If this clause is omitted, the rows are processed in an order the database management system chooses.

ASCENDING/DESCENDING (Optional)

Specifies the order, by column values, in which rows are processed: as low value to high value (ASCENDING) or high value to low value (DESCENDING). ASCENDING is the default and applies until DESCENDING is specified. DESCENDING then remains in effect until ASCENDING is specified again. The effect of ASCENDING/DESCENDING depends on the type of the column value and the database management system.

column

An identifier of a column in the SQL object processed by the FOR. All column identifiers available in an SQL ORDER BY clause are valid (except for the use of an integer representing a column position).

A FOR with an ORDERED BY clause can update the database, unlike an SQL DECLARE CURSOR with an ORDER BY clause. If a FOR with an ORDERED BY clause is needed to update the database, you must define the underlying SQL table with at least one unique index.

You can use the WHERE and ORDERED BY clauses in either order.

statements

PDL statements or SQL statements. The group of statements in the logical scope of a FOR construct can reference or update any column in the row retrieved by the current iteration of the FOR.

You can reference column values for a row only after that row was retrieved by the FOR. You can update column values for a row only in the logical scope of an updateable FOR that processes the row.

WHEN NONE

An optional postscript that specifies that when none of the rows meets the search condition, the statements following the WHEN NONE execute.

WHEN ERROR (Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access the SQLCA and $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR and SQLCA functions are no longer available. For an example of WHEN ERROR processing, see the examples in this section.

Note: Only dataview errors ($ERROR-CLASS=DVW) are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that marks the end of the FOR construct. If FOR statements are nested, the most recent undelimited FOR construct is delimited by the first occurrence of ENDFOR. Each FOR in a nested FOR construct must have a corresponding ENDFOR.

If a row is updated in the logical scope of the FOR construct, the database is updated at the ENDFOR of the current iteration.

Nesting FOR Constructs

You can nest any of the FOR constructs as long as each FOR construct refers to a different dataview. Do not nest a FOR construct for a given dataview in the logical scope of another FOR construct for the same dataview.

When a QUIT is executed in the logical scope of a FOR, the next statement executed is the statement after the ENDFOR. When FOR and LOOP constructs are nested, any construct can be abandoned by referencing the optional label in a QUIT statement. See the QUIT statement in this chapter.

Data Names in FOR Constructs

Columns processed in the FOR construct can be referenced in PDL statements using the column name and, if necessary, the dataview name as qualifier.

For example, column ACCT_NO in dataview ACCT can be compared to the field ACCT_NO in panel PNL1:

```
IF ACCT.ACCT_NO EQ PNL1.ACCT_NO ...
```

You can also use these data items, in addition to working data, parameter data, and panel fields as host variables in embedded SQL. For example, you can use ACCT.ACCT_NO as a host variable in the WHERE clause of an SQL statement.

You cannot make such references before the FOR is executed.

Accessing or Updating Column Values

Statements outside the logical scope of the FOR construct can access, but not update, data from the last row processed for the dataview. The values of the columns processed by the most recent iteration of the FOR are still available after the ENDFOR (see examples in this section), except when the row was deleted or no rows were found (WHEN NONE). This data is available until another FOR accesses the same dataview record.

For example, it might be convenient to first find a row and then refer to its columns rather than nesting the actions in the FOR. You can delegate finding the appropriate row to a lower level procedure.

Updates (changes and deletes) in the logical scope of the FOR. Any update of a column value in the scope of a FOR virtually updates the database. The actual update takes place at the ENDFOR for the current iteration: Therefore, any QUIT or PROCESS NEXT executed in the scope of a FOR abandons the update of the current row even for columns whose values already were changed. A checkpoint in the logical scope of the FOR does not commit the current update because the update does not take place until the ENDFOR. CA Ideal suppresses database writes when it can determine that the database was not altered.

If statements outside the logical scope of the FOR construct attempt to update this data (with a SET, MOVE, and so on), then an execution-time error results.

These rules apply equally to PDL statements and SQL statements. For example, an embedded SQL FETCH statement can update a host variable referencing a column in the logical scope of a FOR construct. The FOR construct must be updateable and access the table containing that column.

Changing the value of a column for the current row in the scope of a FOR construct has no impact on the selection of the next row because selection is made at the time the FOR construct is initially entered.

You cannot use a FOR construct to update an SQL view defined by the database management system as read-only.

Transmit, Checkpoint, or Backout in FOR Constructs (SQL Access)

This note applies to FOR constructs for SQL access that contain a TRANSMIT, CHECKPOINT, or BACKOUT statement, an embedded SQL COMMIT or ROLLBACK statement, or a Debugger breakpoint.

CA Datacom/DB SQL ANSI Mode and DB2

Only a FOR FIRST 1 construct for SQL can contain a TRANSMIT, CHECKPOINT, or BACKOUT statement (or an embedded SQL COMMIT or ROLLBACK statement). If the SQL object is updated, you must define the underlying table with at least one non-nullable, unique index comprised of 64 columns or less.

If any other type of FOR construct for SQL access contains one of these statements, it must quit processing the specified set of rows after the statement is executed because the set is lost at the commit point. For example, an ERROR PROCEDURE invoked from in a FOR EACH can transmit a panel, but it must then QUIT the FOR.

CA Datacom SQL Datacom Mode

Only the BACKOUT statement requires that a QUIT be coded to exit the FOR construct.

Performance Implications

You can improve efficiency by using indexed columns in WHERE and ORDERED BY clauses.

For DB2

As a rule, modify or access only the columns your application needs. Modifying an entire row that was retrieved by a FOR can have significant performance implications. This can happen if you use

- CALL USING UPDATE dataview
- MOVE BY POSITION to the dataview
- MOVE BY NAME to the dataview
- MOVE to a dataview that is an alpha group

In these cases, CA Ideal assumes that all columns are updated, including indexed columns. The database cannot allow access using the index if the indexed column is going to be updated.

SQL Errors

SQL errors and warnings resulting from the execution of either embedded SQL statements or the SQL that CA Ideal generated for a FOR construct are available in the SQL communications area, SQLCA. You can test SQLCA fields for warnings or errors using:

- A copy of the SQLCA defined in working data or parameter data
- $SQL functions
- $ERROR functions

For more information about SQLCA, see the section $SQL Functions (SQL Access Only) in the "Symbolic Debugger Commands" chapter.

If the current iteration of a FOR construct causes an SQL error, control passes to the WHEN ERROR statement at the point of the error. If no WHEN ERROR is coded, control passes to the error procedure.

Listing Generated SQL

To include the SQL generated by the FOR construct in a compiler listing, use the LSQL option on the COMPILE or SET COMPILE command.

Coding for Read-Only Access

If a FOR EACH or FOR FIRST construct is determined to be read-only, regardless of whether NO UPDATE is coded, the FOR FETCH ONLY clause is generated to ensure that the generated SQL performs read-only access.

Example

```
FOR EACH DELINQUENT_ACCT
     WHERE BALANCE > 200
         DO CONTACT_COLLECTOR    : if qualification needed,
ENDFOR                           : use DELINQUENT_ACCT.field
```

Example

```
<<EMP_SEARCH>>
     FOR EACH EMPLOYEE
    WHERE DEPT='D' AND JOB_CODE IN ('J','K','L')
      DO CHECK_GOOD_EMP
      IF ENOUGH_GOOD_EMP
           QUIT EMP_SEARCH
      ENDIF
     ENDFOR
```

Example

```
FOR FIRST INVENTORY_ITEM
     WHERE QOH > 50 AND PRICE BETWEEN 100 AND 500
         DO PROCESS_ITEM
       ENDFOR

FOR THE FIRST 5 INVEN
     WHERE PRICE < COST + 100
          DO P_5_CHEAP_ITEMS
       ENDFOR
```

Example

```
FOR EACH EMPLOYEE
    WHERE DEPT = 'D'
   FOR EACH PAY_REC
       WHERE PAY_REC.EMP_NO = EMPLOYEE.EMP_NO
          DO PROCESS_PAY
   ENDFOR
ENDFOR
```

Example

```
FOR FIRST ACCT
    WHERE PAST_DUE > 90
   ORDERED BY ACCT_NO
        : you can refer to or update "ACCT.field" here
      WHEN NONE
    DO NO_DELINQ_ACCT
       ENDFOR
        : or you can now refer to "ACCT.field" if present
        : you cannot update "ACCT.field" here (unless this
        : is a procedure performed by a DO from in the
        : FOR)

DO FIND_CUSTOMER
       IF CUST_FOUND
         : you can refer to "CUST.field" here
       ELSE
    DO CUST_NOT_FOUND
ENDIF
<<FIND_CUSTOMER>> PROCEDURE
FOR THE CUST
    WHERE CUST_NO = TRANS_CUST_NO
       SET CUST_FOUND = TRUE
WHEN NONE
       SET CUST_FOUND = FALSE
ENDFOR
ENDPROC
```

Example

For CA Datacom SQL access

```
FOR FIRST CUSTOMER
  WHERE CUSTID = PNL-CUST
    DELETE CUSTOMER
WHEN NONE
  NOTIFY 'NO CUSTOMERS FOUND IN' STATE
WHEN ERROR
  SELECT FIRST ACTION
    WHEN $ERROR-DVW-STATUS = -175
       LIST 'Referential Integrity Error: ' $ERROR-CONSTRAINT-NAME
       NOTIFY 'Customer ' CUSTID 'has open orders and cannot be deleted'
   WHEN OTHER
        DO ERROR
  ENDSEL
ENDFOR
```

For DB2

```
FOR FIRST CUSTOMER
  WHERE CUSTID = PNL-CUST
    DELETE CUSTOMER
WHEN NONE
  NOTIFY 'NO CUSTOMERS FOUND IN ' STATE
WHEN ERROR
  SELECT FIRST ACTION
    WHEN $ERROR-DVW-STATUS = -530
       LIST 'Referential Integrity Error: ' $ERROR-CONSTRAINT-NAME
       NOTIFY 'Customer ' CUSTID 'has open orders and cannot be deleted'
   WHEN OTHER
        DO ERROR
  ENDSEL
ENDFOR
```

FOR NEW Statement (SQL Access)

The FOR NEW statement inserts new rows in an SQL object using a dataview defined for the object. The FOR NEW statement is not iterative. To repeat processing of a FOR NEW, you must include it in a looping construct.

This statement has the following format:

```
<<label>>
     FOR [THE] NEW dataview_name
            statements

 [ WHEN DUPLICATE ]
 [     statements ]
 [ WHEN ERROR     ]
 [     statements ]
  ENDFOR
```

<<label>>

An optional 1- to 15-character name of the FOR NEW construct. You can use it to refer to the construct in the QUIT statement.

FOR [THE] NEW

Inserts a new row. You can add the reserved word THE for readability.

FOR NEW initializes the column values in the new row if the program does not initialize them. The column values are initialized to NULL for fields that can have the null value, or to:

- Zeros for numeric fields
- Zero length for variable-length fields
- Spaces for alphanumeric fields
- Current time for time fields
- Current date for date fields
- Current timestamp for timestamp fields

If initial values were specified in the dataview definition, these values initialize the field.

It is recommended that the views you use in FOR NEW include all columns in the underlying table that were defined as NOT NULL. Otherwise, any insert using this view fails because the missing columns cannot be supplied with initial values.

dataview_name

The name of the dataview for which a new row is inserted. The underlying SQL object must be insertable.

Do not qualify the dataview name with an authorization ID.

WHEN DUPLICATE (Optional)

The WHEN DUPLICATE clause contains statements that are executed when the value of an index column to add matches the value of an index existing in the database and when the database does not allow duplicate index values. The criteria for invalid duplicates are defined by the site database administrator when the database is defined.

If you omit the WHEN DUPLICATE clause and duplication is not allowed, control passes to the WHEN ERROR statement if a duplicate is found. If the WHEN ERROR is not coded, control passes to the error procedure.

If duplication is allowed, the WHEN DUPLICATE clause is ignored.

Note: Although the file is not updated when a duplicate record is found (the duplicate record is not added), the WHEN DUPLICATE clause does not affect the execution of the statements that precede it. The statements in the WHEN DUPLICATE clause execute when the duplication is detected at the ENDFOR. At that point, all other statements in the scope of the FOR were already executed. If the FOR construct includes statements that increment counters or set messages, you can correct those values in the WHEN DUPLICATE processing. However, you cannot continue executing the FOR construct.

WHEN ERROR (Optional)

Specifies statements to execute when a database error is encountered in the scope of the FOR construct. If the WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR and $SQL functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR and $SQL functions are no longer available.

Note: Only database errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that terminates the FOR construct. If FOR constructs are nested, the most recent unterminated FOR construct is terminated. You can reference any row added in the FOR NEW after ENDFOR unless a QUIT statement is used.

You can reference columns processed in the FOR construct in PDL statements using the column name and, if necessary, the dataview name as qualifier. Typically, the statements in the scope of the FOR NEW construct place values into the newly created row (see list of examples in this section)

Insertion of a new row into the table or view occurs at the ENDFOR.

Any update of a column value in the scope of a FOR virtually updates the database. The update to the actual table or view takes place at the ENDFOR for the current iteration. Therefore, any QUIT in the scope of a FOR NEW abandons the insertion of the new row. Further reference to that row outside of the FOR is invalid.

You cannot delete a row in the logical scope of a FOR NEW construct.

You cannot reference column values for a row before the FOR is executed.

A FOR NEW construct can contain TRANSMIT, COMMIT, BACKOUT, and so on statements.

SQL errors and warnings, resulting from the execution of either embedded SQL statements or the SQL that CA Ideal generates for a FOR construct are available in the SQL communications area, SQLCA. SQLCA fields can be tested for warnings or errors using:

- A copy of the SQLCA defined in working data or parameter data
- $SQL functions
- $ERROR functions

For more information about SQLCA, see the $SQL Functions (SQL Access Only) in the "Symbolic Debugger Commands" chapter.

If inserting a new row causes the database management system to issue an abnormal error, the WHEN ERROR statements execute. If no WHEN ERROR statement is coded, the error procedure gets control at ENDFOR.

Examples

In this example, one row is added to the INVEN_ITEM table with column values of a numeric literal, an alphanumeric literal, a working data item, and a panel field.

```
FOR THE NEW INVEN_ITEM
    MOVE 1915464 TO CODE
    MOVE 'WIDGETS' TO DESC
    MOVE WORK.COST TO INVEN_ITEM.COST
    MOVE PNL_ASK.QTY TO INVEN_ITEM.QTY
ENDFOR
```

In the following example, the FOR NEW construct is included in a LOOP construct to process multiple records. Notice that a WHEN DUPLICATE clause is specified to correct the NEW_COUNT total when the duplicate record was not added.

```
LOOP UNTIL TRANSCODE = 'Q'
      TRANSMIT INVEN_PNL
      FOR THE NEW INVEN_ITEM
        MOVE INVEN_PNL TO INVEN_ITEM BY NAME
        SET NEW_COUNT = NEW_COUNT + 1
      WHEN DUPLICATE
        SET NEW_COUNT = NEW_COUNT - 1
      ENDFOR
ENDLOOP
```

FOR Statement (Sequential Files)

The FOR statement is used for reading and updating sequential files. The FOR construct begins with a FOR and ends with an ENDFOR.

To retrieve data from a file, you must first define a CA Ideal dataview for the data. The dataview defines the record available to the application.

You can use the following FOR statements for sequential files:

- The FOR EACH, FOR FIRST, and FOR ANY constructs retrieve records. These constructs are iterative. With each iteration, they return the next record in the requested set.
- The FOR NEXT statement accesses a single record from the file.
- Records are added to the file using FOR NEW.

These statements are described in the next section. For more information about CA Ideal dataviews, see the *Creating Dataviews* Guide.

FOR EACH/FIRST/ANY Statement (Sequential Files)

The FOR EACH, FOR FIRST, and FOR ANY statements process a set of records (or process a single record) from a sequential file. All of the statements in the logical scope of the FOR apply to each selected records from the sequential file defined in the specified dataview.

This statement has the following format:

```
[<<label>>]
    [EACH           ]
FOR [ALL            ] dataview_name
    [[THE] FIRST [n] ]
    [ANY n          ]
           [WHERE where condition]
                  statements
           [WHEN NONE     ]
           [    statements]
           [WHEN ERROR    ]
           [    statements]
ENDFOR
```

<<label>>

An optional 1- to 15-character name of the FOR construct. This label refers to the construct in QUIT and PROCESS NEXT statements and as the operand of certain functions such as $COUNT.

EACH|ALL

Indicate that the statements in the scope of the FOR construct apply to every record that satisfies the where condition. You can use the reserved words EACH and ALL interchangeably.

[THE] FIRST [n]

Specifies that the statements in the scope of the FOR construct apply to the first n records that satisfy the where condition. The value specified for n can be an identifier of a numeric field or a numeric literal that specifies the number of records to process. The default is FIRST 1. You can add the reserved word THE for readability.

ANY n

Specifies that the statements in the scope of the FOR construct apply to any n records that satisfy the where condition. The value specified for n can be an identifier of a numeric field or a numeric literal that specifies the number of records to process. N is required for the FOR ANY clause.

For sequential dataviews, ANY n is equivalent to FIRST n.

dataview-name

The name of the dataview processed.

- **WHERE clause**(Optional)

  Specifies that the statements in the scope of the FOR construct apply to those records that satisfy the condition.

  where-condition

  A condition with the following qualifications:

  - If the condition is a condition name, it must be from the dataview referenced.
  - The left-hand operand of each relational-expression must be the identifier of a field or alpha-group in the dataview referenced.
  - If the left-hand operand is an alphanumeric field, the right-hand operand must be an alphanumeric expression.
  - If the left-hand operand is a numeric field, the right-hand operand can be a numeric expression, an alphanumeric expression, or a non-alpha group that is not a panel group or dynamic matching parameter. When the right-hand operand is not a numeric expression, a warning is issued when the program is compiled and, if the right-hand operand cannot be converted to numeric, a runtime error occurs.
  - The left-hand operand of a relational-expression in a where condition need not be qualified as to dataview name since it refers implicitly to the dataview in the FOR clause. However, reserved words used as operands must always be qualified.
  - The right-hand operand of a relational-expression can be any arithmetic or alphanumeric expression, but cannot reference any fields in the dataview named in the FOR clause.
  - Simple conditions cannot be Boolean functions or flags.
  - Any subscripts used in the where condition must not be numeric fields in the dataview referenced.

statements

PDL statements. The group of statements in the logical scope of a FOR construct can reference any field in the record processed by the FOR.

WHEN NONE

An optional postscript that specifies that, when none of the records meets the where condition, the statements following the WHEN NONE execute.

WHEN ERROR (Optional)

Specifies statements to be executed when a dataview error is encountered in the scope of the FOR construct. If the WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that marks the end of the FOR construct. If FOR statements are nested, the most recent unterminated FOR construct is terminated by the first occurrence of ENDFOR. Each FOR in a nested FOR construct must have a corresponding ENDFOR.

You cannot read sequential files online under CICS.

You cannot modify sequential files in the scope of a FOR EACH construct, even if the dataview is marked updateable. Sequential files are updated by writing records to a new file.

The keyword QUIT in the logical scope of a FOR EACH abandons processing of the set of records. The next statement executed is the statement after the ENDFOR. When FOR and LOOP constructs are nested, you can abandon any construct by referencing the optional label in a QUIT statement. See the QUIT statement in this chapter.

You can reference fields processed by each iteration of the FOR construct in PDL statements. The identifier is the name of the field defined in the dataview or the field name with the dataview name as qualifier. For example, you can compare field ACCT_NO in dataview ACCT to the field ACCT_NO in panel PNL1:

```
IF ACCT.ACCT_NO EQ PNL1.ACCT_NO ...
```

You cannot make such references before the FOR executes.

Sometimes it is convenient to first process a dataview record and then refer to its fields rather than to code the actions in the FOR. For example, you can delegate finding the appropriate record to a lower level procedure.

You can nest any of the FOR constructs as long as each FOR construct refers to a different dataview. Do not nest a FOR construct for a given dataview in another FOR construct for the same dataview.

FOR NEXT Statement (Sequential Files)

The FOR NEXT statement specifies a series of statements that apply only to the next record of a sequential file dataview. If a previous FOR FIRST was executed for the same dataview, the next record in sequence is accessed. If no previous FOR FIRST was executed, FOR NEXT accesses the first record.

Note: This statement is not iterative. To repeat execution of this statement, you must code it in a LOOP construct.

Under z/OS or VSE, this construct applies only to sequential file dataviews in applications run in batch since sequential files cannot be read online.

This statement has the following format:

```
<<label>>
     FOR [THE] NEXT dataview_name
            statements

   [ WHEN NONE      ]
   [     statements ]

   [ WHEN ERROR     ]
   [     statements ]

  ENDFOR
```

<<label>> (Optional)

Specifies a 1- to 15-character name of the FOR construct. You can use it to refer to the construct in a QUIT statement.

FOR [THE] NEXT

Specifies that the action to take only applies to the next record of a sequential file dataview. You can add the reserved word THE for readability.

dataview-name

Specifies the name of the dataview that defines each record to read.

WHEN NONE(Optional)

Reserved words that specify that when no next record exists (for example, there are no more records in the file), the statements following the WHEN NONE execute.

WHEN ERROR(Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that terminates the FOR construct. If FOR constructs are nested, the most recent unterminated FOR construct is terminated by the first occurrence of ENDFOR. Each FOR must have a corresponding ENDFOR.

The FOR EACH construct is used for most sequential processing. The FOR NEXT construct is used in situations where only one record, following the current record, is required.

You can nest any of the FOR constructs as long as each FOR construct refers to a different dataview. Do not nest a FOR construct for a given dataview in another FOR construct for the same dataview.

Example

```
FOR NEXT EMPLOYEE
   IF STATUS = 'T'
     SET MSG = 'EMPLOYEE TERMINATED'
  ENDIF
ENDFOR
```

Example

```
<<MAIN>>PROCEDURE

<<LOAD>> LOOP
  FOR NEXT STUDENT_QSAM
    IF STUDENT_QSAM.CUM_GPA >= 3.5
       PRODUCE JEDEANS
    ENDIF
  WHEN NONE
    IF $COUNT (LOAD) = 0
        LIST 'NO RECORDS IN STUDENT_QSAM'
    ELSE
        SET RECORDS = $COUNT (LOAD)
       LIST RECORDS 'RECORDS PROCESSED ' SKIP
        LIST '***END OF FILE ***'
   ENDIF
    QUIT RUN
  ENDFOR

  FOR NEW STUDENT_LOAD
    MOVE STUDENT_QSAM TO STUDENT_LOAD BY NAME
  WHEN DUPLICATE
    LIST 'STUDENT #'
    STUDENT_QSAM.STUDENT.NR 'ALREADY EXISTS'
 ENDFOR
ENDLOOP
```

FOR NEW Statement (Sequential Files)

The FOR NEW statement adds new records to the end of a sequential file using a dataview for that file. The FOR NEW statement is not iterative. To repeat processing of a FOR NEW, you must include it in a looping construct.

This statement has the following format:

```
[<<label>>]
FOR [THE] NEW dataview-name
     statements

    [ WHEN ERROR     ]
    [     statements ]
ENFOR
```

<<label>>

An optional 1- to 15-character name of the FOR NEW construct. You can use it to refer to the construct in the QUIT statement.

FOR [THE] NEW

Specifies the action to take to add each new record.

FOR NEW initializes the values in the record defined by the specified dataview. If initial values were specified for the field in the dataview definition, they are used. Otherwise, the fields are initialized to zeros (for numeric fields) and blanks (for alphanumeric fields).

Note for the DBA: Fields in the underlying record that are not defined in the modeled dataview are initialized to spaces, without regard to the intended data type of the field. Therefore, it is recommended that dataviews used in FOR NEW span the entire record.

You can add the reserved word, THE, for readability.

dataview-name

The name of the dataview for which a new record is added. The dataview must be updateable.

statements

PDL statements. Typically, the statements in the scope of the FOR NEW construct are those that place values into the newly created record.

WHEN ERROR (Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that terminates the FOR construct. If FOR constructs are nested, the most recent unterminated FOR construct is terminated. You can reference any recently added field in the dataview record after ENDFOR unless a QUIT statement is used.

The file is opened when the FOR NEW statement executes. The record is actually inserted at the ENDFOR.

A QUIT in the logical scope of a FOR NEW aborts the creation of the record, that is, creates an empty sequential file.

PROCESS NEXT has no meaning in the FOR NEW.

Dataview fields cannot be referenced before execution of the FOR statement for the dataview.

The first time a FOR NEW is executed, the file is opened for output and the disposition parameter takes control to determine where the record is added. In a z/OS environment, DISP=MOD, the new record is added to the end of the file. If DISP=OLD, the existing records are purged and the new record is added to the beginning of the file. Subsequent records are added to the end of the file in both cases. See the following examples.

Online, if one or more users are writing to the file, new records are interleaved. If a sequential file is written online under CICS, you cannot read it in batch until CICS closes it.

Example

```
FOR THE NEW INVEN_ITEM
      MOVE TRANS_INFO TO INVEN_ITEM BY NAME
ENDFOR
```

The following table illustrates the differences in the processing of a sequential file when the disposition parameter is set to OLD or MOD in z/OS.

Statements	Result DISP=OLD	DISP=MOD
FOR EACH dvwname statements ENDFOR	Reads existing records	Reads existing records
LOOP 3 TIMES FOR NEW dvwname statements to add three new records ENFOR ENDLOOP	Purges existing records and adds three new records to beginning of file	Peeps original records and adds three new records to end of file
FOR EACH dvwname statements ENDFOR	Reads three new records that were just added	Reads original records and three newly added records
LOOP 2 TIMES FOR NEW dvwname statements to add 2 new records ENDFOR	Purges three newly added records and adds two new records	Keeps original records and three added records and adds two new records to the end of the file
FOR EACH dvwname . . .	Reads only the two most recently added records	Reads original records plus five new records

FOR Statement (VSAM Files)

The FOR statement is used for reading and updating files. The FOR construct begins with a FOR and ends with an ENDFOR.

Note: To access VSAM files from CA Ideal, your site must have the CA Ideal VSAM support option installed.

To process data from a VSAM file, you must first define the data in a CA Ideal dataview and catalog it. The dataview defines the field names that you can use in the CA Ideal application. CA Ideal supports dataviews for all three types of VSAM files:

- KSDS-Key Sequenced Data Sets
- ESDS-Entry-Sequenced Data Sets
- RRDS-Relative Record Data Sets

The record length in a VSAM file can vary in two ways:

- Variable-occurrence records include a field or group of fields that repeats a specified number of times. These records are defined using an OCCURS DEPENDING ON clause to indicate which field controls the number of times the repeating field or group of fields occurs in the individual record.
- Variable-segment records include different fields, based on a record type that is usually included in a fixed portion of the record. The varying sets of fields (segments) are defined using the REDEFINES clause to establish different level-2 field descriptions that overlap each other. These varying segments can be different lengths, but the longest segment must be the first variable segment defined.

ESDS files are not required to have a fixed-length segment. KSDS files require a fixed-length segment, since the keys must always be in the same position in the record. For RRDS files, variable-occurrence records are not supported. Variable-segment records are accommodated to allow the use of multiple record types; however, the actual records written are padded with binary zeros, as required, to create fixed-length records.

The FOR EACH and FOR FIRST constructs retrieve and update one record at a time. These constructs are iterative; with each iteration, they return the next requested record. It is not necessary to create an image of the record in working data. CA Ideal maintains a data structure that contains the record retrieved by each iteration of the FOR. PDL statements can use the fields in this structure. If a record is updated in the scope of the FOR (and updating is allowed), the data set is automatically updated at the ENDFOR for the current iteration.

To insert a new record into the file, use the FOR NEW statement. The FOR NEW statement adds a single record that is not iterative. To repeat processing of a FOR NEW statement, place the statement in a looping structure, such as a LOOP construct.

These FOR statements are described in the next sections. For information on creating, cataloging, and displaying VSAM dataviews, see the *Creating Dataviews Guide*.

FOR EACH/FIRST Statement (VSAM Files)

The FOR EACH and FOR FIRST statements process a set of records (or a single record) from a VSAM file. The FOR construct is iterative. With each iteration, it returns the next record in the requested set. All of the statements in the scope of the FOR apply to each record selected. FOR FIRST and FOR EACH are the only constructs that update or delete a record.

This statement has the following format:

```
<<label>>
    [EACH              ]
FOR [ALL               ]   dataview_name[NO UPDATE]
    [[THE] FIRST [n] ]
      [WHERE where-condition ]
      [ORDERED BY  [ASCENDING ] [FIELD-NAME ]]
      [            [DESCENDING]              ]
               statements
        [WHEN NONE     ]
        [    statements]
        [WHEN ERROR     ]
        [     statements]
ENDFOR
```

<<label>>

An optional 1- to 15-character name of the FOR construct. You can use this label to refer to the construct in QUIT and PROCESS NEXT statements and as the operand of certain functions such as $COUNT.

EACH|ALL

Indicate that the statements in the scope of the FOR construct apply to every record that satisfies the where condition. You can use the reserved words EACH and ALL interchangeably.

[THE] FIRST [n] (Default)

Specifies that the statements in the scope of the FOR construct apply to the first n records that satisfy the where condition. The value specified for n can be a numeric literal or the name of a numeric field. The default is FIRST 1. You can add the reserved word THE for readability. When you use FOR FIRST n with a where condition and an ORDERED BY clause, all records that satisfy the where condition are ordered and then the first n ordered records are selected.

dataview-name

Specifies the name of the VSAM dataview.

NO UPDATE (Optional)

Specifies that the records processed by this FOR construct are not updated. This applies even if the dataview is defined as updateable (Update Intent = Y in the parameter definition). If used, this clause must immediately follow the dataview name. The use of this clause for browsing provides significant efficiency gains.

WHERE clause (Optional)

Specifies that the statements in the scope of the FOR construct apply to those records that satisfy the specified condition. If the WHERE clause is omitted, all records in the data set are processed. The WHERE clause can also determine the index where the data set is accessed (see the description of the ORDERED BY clause for more detailed information).

where-condition

A condition (as defined in the beginning of chapter 2) with the following further qualifications:

- The left-hand operand of each relational-expression must be the identifier of a field or group in the dataview referenced or the function $RBA (for an ESDS data set) or $RRN (for an RRDS data set).
- If the left-hand operand is an alphanumeric field, the right-hand operand must be an alphanumeric expression.
- If the left-hand operand is a numeric field, the right-hand operand can be a numeric expression, an alphanumeric expression, or a non-alpha group that is not a panel group or dynamic matching parameter. When the right-hand operand is not a numeric expression, a warning is issued when the program is compiled and, if the right-hand operand cannot be converted to numeric, a run-time error occurs.
- If the left-hand operand is the function $RRN, the value of the right-hand operand must be greater than or equal to one. If the left-hand operand is the function $RBA, the value of the right-hand operand must be greater than or equal to zero.

- A field name used as the left-hand operand of a relational-expression in a where condition does not need to be qualified with a dataview name since it refers implicitly to the dataview in the FOR clause. However, reserved words used as operands must always be qualified. The $RBA and $RRN functions must not include the dataview name.
- The right-hand operand of a relational-expression can be any arithmetic or alphanumeric expression, but cannot reference any fields in the dataview named in the FOR clause. In particular, subscripts that qualify dataview array elements cannot depend on data in the dataview.
- If the condition is a condition name (a type C field), it must be from the dataview being referenced.
- Simple conditions cannot be Boolean functions or flags.

ORDERED BY clause (Optional)

Identifies the index by which a KSDS data set is accessed and determines the logical order in which the records in any VSAM file are processed. If this clause is omitted, the index is determined as follows:

- If a WHERE clause is specified that uses only one key (full or high-order partial), that key identifies the index to use.
- If neither an ORDERED BY nor a WHERE clause is specified, the primary key is used.

If the ORDERED BY clause is omitted and a WHERE clause is specified that uses more than one key, an error message is issued when the program is compiled.

ASCENDING/ DESCENDING (Optional)

Specifies whether processing proceeds from low to high values (ASCENDING) or high to low values (DESCENDING). ASCENDING is the default. The effect of ASCENDING/DESCENDING depends on the type of the VSAM data set:

ESDS data set

Records are retrieved moving through the file in a forward (ASCENDING) or backward (DESCENDING) direction, according to the relative byte address (RBA). You cannot specify a field-name to determine the order.

RRDS data set

Records are retrieved according to the relative record number, moving from the lowest number to the highest (ASCENDING) or from the highest number to the lowest (DESCENDING). You cannot specify a field-name to determine the order.

KSDS data set

Records are retrieved according to ascending or descending values in the field specified as field-name. Records are retrieved in collating sequence. If the specified field-name identifies a signed numeric (type N) field, the collating sequence cannot return the records in the expected algebraic order.

field-name

The name of a field or group of fields defined as a primary or alternate key. You can only specify the field-name for a KSDS file. You can specify only one field-name. It cannot be subscripted.

statements

PDL statements. The statements in the logical scope of a FOR construct can reference any field in the record most recently processed by the FOR. However, it is the programmer's responsibility to check the record type before referencing any fields in a variable-segment record.

WHEN NONE

An optional postscript that specifies that when none of the records meets the where condition, the statements following the WHEN NONE execute.

WHEN ERROR(Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that marks the end of the FOR construct. If FOR statements are nested, the most recent undelimited FOR construct is delimited by the first occurrence of ENDFOR. Each FOR in a nested FOR construct must have a corresponding ENDFOR.

Updates are actually written to the file at the ENDFOR of the current iteration of the FOR construct.

You can reference fields processed by each iteration of the FOR construct in PDL statements. The identifier is the field name or the field name with the dataview name as qualifier. For example, the field ACCT-NO in the ACCT dataview can be compared to the field ACCT-NO in the panel named PNL1:

```
IF ACCT.ACCT-NO EQ PNL1.ACCT-NO ...
```

You cannot make such references before the FOR executes.

Sometimes it is convenient to first process a dataview record and then refers to its fields rather than coding the actions in the FOR. For example, you can delegate finding the appropriate record to a lower level procedure, as shown in following examples in this section.

Statements outside the logical scope of the FOR construct can access, but not update, data in the dataview record. Data in the last record processed is available after the ENDFOR until another FOR accesses the same dataview, except when the record was deleted or no records were found (WHEN NONE).

The access key used to retrieve records should not be updated in the scope of a FOR statement based on that key. If the access key is the primary key, VSAM does not allow you to change it. If the access key is an alternate key, you can update it in CA Ideal; however the results can be unpredictable:

- If you update the alternate key and the index is in the upgrade set, the index changes dynamically while you are scanning the file.
- If the index is not in the upgrade set, the changes do not reflect in the index. Whenever you use an index that is not part of the upgrade set, results can be inaccurate.

You must make updates (changes and deletes) in the logical scope of the FOR. Any update of a dataview field in the scope of a FOR logically updates the file. For changes (but not for deletes), the actual update takes place at the ENDFOR for the current iteration. Therefore, any QUIT or PROCESS NEXT executed in the scope of a FOR abandons the update of the current record even for fields whose values already were changed. A checkpoint in the logical scope of the FOR does not commit the current update, because the update does not take place until the ENDFOR.

VSAM records are updated only through the primary index. Before updating a record, CA Ideal rereads the record with exclusive control, using the primary key. CA Ideal can then verify that the record was not deleted or changed since the original access before updating or deleting the record.

During updates, CA Ideal writes a record only when it can determine that the data was altered. CA Ideal always processes CHECKPOINT and BACKOUT requests, even if no actual changes were made to the file. In a CICS environment, the CHECKPOINT and BACKOUT statements execute the CICS SYNCPOINT and SYNCPOINT ROLLBACK commands. In non-CICS environments, the BACKOUT statement has no effect. The CHECKPOINT statement performs a TCLOSE operation that flushes certain VSAM buffers if there was any VSAM access before the CHECKPOINT.

If statements outside the logical scope of the FOR construct attempt to update the record (with a SET, MOVE, and so on), a run-time error results.

- Access the file with a unique key when updating records since the execution of the FOR EACH or FOR FIRST *n* statements cannot resume if the access key is non-unique and any of the following occurs in the scope of the FOR statement:
- TRANSMIT
- Update of the data set named in the FOR statement
- CHECKPOINT
- BACKOUT
- Debugger breakpoint

Make sure that the above actions do not occur in procedures or subprograms called from in the FOR construct.

In the scope of a FOR statement that accesses a VSAM data set, a non-ideal subprogram should not access that VSAM data set or issue a CICS SYNCPOINT.

The FOR EACH and FOR FIRST statements read the set of records that satisfies the conditions specified in the WHERE clause and then process those records according to the statements entered in the FOR construct. If the WHERE clause is complex, the set of records to read is determined by establishing a search interval based on the conditions in the WHERE clause.

Conditions connected by an OR can result in an overly large search interval. For example, if the WHERE clause specifies the highest value of a key field and the lowest value of a key field, connected by an OR, the search interval includes the entire file.

You can nest any of the FOR constructs as long as each FOR construct refers to a different dataview. Do not nest a FOR construct for a given dataview in another FOR construct for the same dataview.

If more than one record in a file is needed simultaneously, either use two dataviews for the same file or save necessary information in working data.

When a QUIT is executed in the logical scope of a FOR, the next statement executed is the statement after the ENDFOR. When FOR and LOOP constructs are nested, any construct can be abandoned by referencing the optional label in a QUIT statement. See the QUIT statement in this chapter.

You can use the WHERE and ORDERED BY clauses in either order.

You cannot delete records from an ESDS file; therefore, you cannot specify the DELETE statement in a FOR construct that accesses an ESDS file.

When updating an ESDS data set, you cannot change the length of a record. With CA Ideal, you can decrease the logical length of a record (either by reducing the number of occurrences in a variable-occurrence record or by changing the record to a shorter variable-segment record with the SET $REC-SEGMENT command), but the remaining length of the updated record is padded with binary zeros. Do not use this technique when record-types are identified by record-length since the original length is retrieved when the record is read.

You can improve efficiency in the following ways:

- Use an ORDERED BY clause.
- Use a single, complete key as a group field in the WHERE clause.
- If both an ORDERED BY and a WHERE clause are present and reference key fields, use the same key in both clauses.
- Do not use different keys or non-key fields in WHERE clause conditions connected by OR. Even using the same key can read a significant number of extra records if the key values are widely divergent. In the latter case, you can use multiple, unnested FOR constructs to retrieve only the appropriate records (see following examples in this section).
- Use alphanumeric (type X) fields whenever possible as key fields.

Examples

In the following example, assume that the BALANCE field is defined as an alternate key for the DELINQUENT-ACCT dataview. Since the BALANCE field is used in the WHERE clause and there is no ORDERED BY clause, the BALANCE field is used as the access key.

```
FOR EACH DELINQUENT-ACCT
    WHERE BALANCE > 200
      DO CONTACT-COLLECTOR
ENDFOR
```

In the following example, the access key is determined by the ORDERED BY clause since neither the QOH nor the PRICE field are key fields.

```
FOR FIRST INVENTORY-ITEM
    WHERE QOH > 50 AND PRICE < 500
    ORDERED BY ITEM-NAME
        DO PROCESS-ITEM
ENDFOR
```

For the INVEN dataview used in the following example, the PRICE field is defined as an alternate key and is used as the access key, based on its use in the WHERE clause. Since PRICE is not a unique key, if the P-5-CHEAP-ITEMS procedure includes an update or a TRANSMIT statement, processing cannot resume after the first iteration of the FOR and an error occurs.

```
FOR THE FIRST 5 INVEN
    WHERE PRICE < 100
        DO P-5-CHEAP-ITEMS
ENDFOR
```

In the following example, the EMPLOYEE file is accessed using the unique key EMP-NO; however, each record in the file is read in sequence to find the records that satisfy the WHERE clause, since the key field EMP-NO is not specified in the WHERE clause. If EMP-NO in the PAY-REC dataview is also a key field, the PAY-REC dataview is accessed randomly (only the record that matches the EMPLOYEE record is read).

```
FOR EACH EMPLOYEE
    ORDERED BY EMP-NO
        WHERE DEPT = 'D'
            FOR EACH PAY-REC
                WHERE PAY-REC.EMP-NO = EMPLOYEE.EMP-NO
                   DO PROCESS-PAY
            ENDFOR
ENDFOR
```

The following example shows the use of the WHEN NONE clause and indicates when you can access and update dataview fields.

```
FOR FIRST ACCT
   WHERE PAST-DUE > 90
      ORDERED BY ACCT-NO
                 : you can refer to or update ACCT.field here
         WHEN NONE
      DO NO-DELINQ-ACCT
       ENDFOR
    : you can now refer to ACCT.field if present
    : you cannot update ACCT.field here (unless this
    : is a procedure performed by a DO statement
    : in the FOR, but not in the WHEN NONE)
```

The following example shows the use of FOR statement is included in the procedure FIND-CUSTOMER. You cannot update the record in the subsequent IF statement, although you can reference, display, or list the fields.

```
DO FIND-CUSTOMER
       IF CUST-FOUND
                    : you can refer to "CUST.field" here
ELSE
    DO CUST-NOT-FOUND
ENDIF

<<FIND-CUSTOMER>> PROCEDURE

   FOR THE FIRST CUST
      WHERE CUST-NO = TRANS-CUST-NO
         SET CUST-FOUND = TRUE
      WHEN NONE
         SET CUST-FOUND = FALSE   ENDFOR
```

In the following example, the PSS-MASTER dataview is accessed by relative byte address using the $RBA function to determine the starting record. The dataview name is not specified for the function when it is used in the WHERE clause.

```
FOR FIRST 21 PSS-MASTER
    WHERE $RBA GE START-ADDRESS  :function is not qualified
        statements
ENDFOR
SET LAST-ADDRESS = $RBA(PSS-MASTER)      :function is qualified
```

In the following example, two FOR constructs were specified to retrieve the records that satisfy the condition EMP-ID < 101 OR EMP-ID > 846. In this example, the PSS-MASTER file is a KSDS file with the key field EMP-ID.

```
FOR EACH PSS-MASTER
   WHERE EMP-ID < 101
       statements
ENDFOR
FOR EACH PSS-MASTER
   WHERE EMP-ID > 846
       statements
ENDFOR
```

With the FOR EACH statement, the results of the two FOR statements above are the same as that for a single FOR EACH statement with the combined condition WHERE EMP-ID < 101 OR EMP-ID > 846. However, with a FOR FIRST statement, the results might not be the same since, with the single construct, only the first record that satisfies the combined condition is retrieved, but with a double construct, two records are retrieved.

In the following example, two FOR constructs access a weekly transaction file called TRAN-FILE. The FOR FIRST construct loads an RBA table from the first record that contains the RBAs used for each day's transactions. The FOR EACH construct retrieves Tuesday's records by comparing the $RBA function to the RBA table values for Tuesday and Wednesday.

```
FOR FIRST TRAN-FILE
   SET W-RBA-TABLE = TRAN-FILE.TABLE-SEG BY NAME
ENDFOR
statements
FOR EACH TRAN-FILE
   WHERE $RBA GE W-RBA-TABLE.TUESDAY
      AND $RBA LT W-RBA-TABLE.WEDNESDAY + 1
          statements
ENDFOR
```

FOR NEW Statement (VSAM Files)

The FOR NEW statement inserts a single record into a VSAM file using a dataview defined for the file. The FOR NEW statement is not iterative. To repeat processing of a FOR NEW, you must include it in a looping construct.

This statement has the following format:

```
<<label>>
   FOR [THE] NEW dataview-name
       [WHERE $RRN = value]
           statements
       [WHEN DUPLICATE]
       [    statements]
       [WHEN ERROR    ]
       [    statements]
   ENDFOR
```

<<label>>

An optional 1- to 15-character name of the FOR NEW construct. You can use it to refer to the construct in a QUIT statement.

FOR [THE] NEW

Specifies the action to take to insert or add each new record. You can add the reserved word, THE, for readability.

FOR NEW initializes the field values in the new record if the program did not initialize them previously. If initial values were specified for the field in the dataview definition, they are used. Otherwise the fields are initialized to zeros (for numeric fields) and blanks (for alphanumeric fields).

dataview-name

The name of the dataview that defines the new record inserted. The dataview must be updateable.

WHERE $RRN = value

Specifies the relative record number of the new record in an RRDS VSAM data set. The operator must be the equal sign (=). This statement is required for RRDS files, but cannot be used for ESDS files or KSDS files.

value

- Can be an integer number, a numeric expression, or the name of a numeric field or data item. The value must be greater than or equal to one.

statements

PDL statements. Typically, the statements in the scope of the FOR NEW construct are those that place values into the newly created record.

WHEN DUPLICATE(Optional)

Used only for KSDS and RRDS data sets.) The WHEN DUPLICATE clause contains statements that execute when any key value of a record to add matches the key value of a record existing in the file and the index is defined as unique. For RRDS files, the record is a duplicate when the relative record number matches the relative record number of a record in the file.

If the WHEN DUPLICATE clause is omitted and the index is defined as unique, a run-time error occurs and control passes to the WHEN ERROR statement if a duplicate record is encountered. If the WHEN ERROR is not coded, control passes to the error procedure.

If the WHEN DUPLICATE clause is included and the index is not defined as unique, the WHEN DUPLICATE clause is ignored. The WHEN DUPLICATE clause is also ignored when it is specified for an ESDS file.

Note: Although the file is not updated when a duplicate record is found (the duplicate record is not added), the WHEN DUPLICATE clause does not affect the execution of the statements that precede it. The statements in the WHEN DUPLICATE clause execute when the duplication is detected at the ENDFOR. At that point, all other statements in the scope of the FOR were already executed. If the FOR construct includes statements that increment counters or set messages, you can correct those values in the WHEN DUPLICATE processing. However, you cannot continue executing the FOR construct.

WHEN ERROR (Optional)

Specifies statements to execute when a dataview error is encountered in the scope of the FOR construct. If WHEN ERROR is not specified, errors are processed by the user-defined or default error procedure.

The statements specified following a WHEN ERROR clause can access $ERROR functions and should resolve the error with either a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

Note: Only dataview errors are handled by the WHEN ERROR clause. System and internal errors are handled by the user-specified or default error procedure.

ENDFOR

A reserved word that terminates the FOR construct. If FOR constructs are nested, the most recent unterminated FOR construct is terminated. You can reference any field in the dataview record just added after ENDFOR unless a QUIT statement is used.

A QUIT or PROCESS NEXT in the logical scope of a FOR NEW abandons the creation of the record. Further reference to fields in the dataview outside of the FOR is invalid.

Insertion of a new record into the file occurs at the ENDFOR.

You cannot delete a record in the dataview specified in the FOR NEW construct in the logical scope of the FOR NEW construct.

Examples

In this example, the FOR NEW construct is included in a LOOP construct to process multiple records. Notice that a WHEN DUPLICATE clause is specified to correct the NEW-COUNT total when the duplicate record was not added.

```
LOOP UNTIL TRANSCODE = 'Q'
   TRANSMIT INVEN-PNL
   FOR THE NEW INVEN-ITEM
      MOVE INVEN-PNL TO INVEN-ITEM BY NAME
      SET NEW-COUNT = NEW-COUNT + 1
      WHEN DUPLICATE
         SET NEW-COUNT = NEW-COUNT - 1
   ENDFOR
ENDLOOP
```

In this example, the $RRN function determines the location of the new record. First, the file is read in descending order to determine the relative record number of the last existing record. Then the new records are added, starting with the next relative record number.

```
FOR FIRST PAYROLL ORDERED BY DESCENDING
    SET NEXT-REC = $RRN(PAYROLL) + 1
  WHEN NONE
    SET NEXT-REC = 1
ENDFOR
LOOP UNTIL DONE
   FOR NEW PAYROLL
       WHERE $RRN = NEXT-REC
          DO NEW-SETUP
          SET NEXT-REC = $RRN(PAYROLL) + 1
          MOVE NEW-PAY TO PAYROLL BY NAME
   ENDFOR
ENDLOOP
```

VSAM Support: Backout and Recovery

CA Ideal supports VSAM file access through the FOR construct. As for any other data access method, you want to ensure the integrity of VSAM files in the event of system failures, disk problems, and so on. Following is a description of backout and recovery for VSAM files that CA Ideal accesses.

In non-CICS environments, the CA Ideal PDL CHECKPOINT statement results in a CLOSE (Type=T) operation in z/OS and a TCLOSE operation in VSE. The ROLLBACK statement has no effect.

When VSAM is invoked under CICS, CICS command level functions are actually used to access VSAM files. CICS provides the facilities that ensure the integrity of VSAM files. If these facilities are installed and enabled for VSAM files accessed in a CA Ideal program, the Procedure Definition statements CHECKPOINT and BACKOUT function as expected for VSAM files: A CHECKPOINT results in a CICS SYNCPOINT and a BACKOUT statement results in a SYNCPOINT ROLLBACK.

In addition, if CICS VSAM files are accessed along with CA Datacom/DB and DB2, the CICS SYNCPOINT and SYNCPOINT ROLLBACK facilities help synchronize VSAM files with the other databases (each of which has its own recovery facilities).

IF Statement

The IF statement chooses one of two alternative courses of action, depending on whether a condition is true, false, or unknown.

This statement has the following format:

```
IF condition

[THEN         ]
[   statements]
[ELSE        ]
[   statement]
ENDIF
```

condition

A user-defined condition (see the definition of a condition in chapter one). Statements that immediately follow the IF [THEN] condition are executed only if the condition is true.

THEN

A reserved word that you can add for readability.

ELSE

Marks the start of a set of statements to execute if the condition is False or Unknown. If you omit ELSE and the condition is False or Unknown, the IF statement does not cause any action and the next statement after the ENDIF executes.

ENDIF

Terminates the IF construct. When IF statements are nested, the most recent unterminated IF construct is terminated by the first occurrence of ENDIF. Each IF in a nested IF construct must have a corresponding ENDIF.

Examples

```
IF QUANT_ON_HAND > QUANT_ORDERED
   SUBTRACT QUANT_ORDERED
     FROM QUANT_ON_HAND
ELSE
MOVE "OUT OF STOCK" TO MESSAGE
   PRODUCE EX_LINE
ENDIF

IF NOT SUFF_ON_HAND
    DO REORDER_ITEM
ENDIF

IF (EMP_DEPT = 'D' AND JOB_CODE = 'J')
   OR RECENTLY_HIRED
      DO PROCESS_JUNIOR
ENDIF
```

```
<<MAIN>> PROCEDURE

FOR EACH PAYROLL
   WHERE YTD_COMMISSION > 7500
      FOR EMPLOYEE
         WHERE NUMBER = PAYROLL.NUMBER
         SET W_YTD_NET = (YTD_WAGES + YTD_COMMISSION)
         IF ACTIVITY_CODE = 'A'                      :ACTIVE
            IF ACTIVITY_STATUS = 'S'                 :S=SALARIED
               SET W_TAG = 'SALARIED'
            ELSE                                     :H=HOURLY
               SET W_TAG = 'HOURLY'
            ENDIF
         ELSE                                        :INACTIVE
            SET W_TAG = 'INACTIVE'
         ENDIF
      LIST EMPLOYEE.NUMBER EMPLOYEE.NAME
         W_YTD_NET  W_TAG
      WHEN NONE
      ENDFOR
ENDFOR
ENDPROC
```

INITIATE Statement

The INITIATE statement runs a CA Ideal program asynchronously online.

This statement has the following format:

```
INITIATE pgm-name [USING INPUT parm]
```

pgm-name

The name of the CA Ideal program to execute as an asynchronous task. The specified program must be a valid resource of the program that contains this statement and must not contain a TRANSMIT statement. A non-ideal subprogram cannot be invoked by INITIATE, although it can be CALLed by the CA Ideal program that was INITIATEd.

parm

The name of a field that contains data or a literal to pass as an input-only parameter to the program specified as pgm-name. Only one parameter can be passed to pgm-name and that parameter cannot be a group field.

The asynchronous task uses the same selected system as the current run. Therefore, the initiated program must be in the same system as the main program executed by the RUN command. When the asynchronous subprogram completes, a message is sent to the initiating session indicating that the asynchronous task completed. You can use the SET ASYNCMSG command to suppress this message.

Example

The following code is part of an inventory update procedure. On the INCOMING panel where the user enters the data on the incoming shipments of stocked items (some of which were backordered), the PF5 key is defined to print a report of all outstanding orders for a back-ordered product. With the INITIATE statement, you can produce this report as an asynchronous task so you can proceed to other tasks online.

```
<<PROCESS-INCOMING>> PROCEDURE
   TRANSMIT INCOMING
   SELECT FIRST ACTION
      WHEN $PF3
         QUIT PROCEDURE
      WHEN $PF5
         DO UPDATE-ITEM
         INITIATE PRTORDER USING INPUT INCOMING.ITEM-ID
      WHEN $ENTER-KEY
         DO UPDATE-ITEM
      WHEN NONE
         DO ERROR
   ENDSEL
ENDPROC
```

The PRTORDER program uses the ITEM-ID entered with the other inventory data on the panel INCOMING to find the orders that included that item.

INVERT Statement

The INVERT statement reverses the order of characters in a given alphanumeric field or alphanumeric group.

This statement has the following format:

```
INVERT {alpha_field | alpha_group}
```

Example

The following example shows how you can use INVERT in text processing to find the start of the last word in a sentence. W-WORK is defined as type V for a variable length field.

```
SET W_SENTENCE = 'THIS IS A SENTENCE
'SET W_WORK = $TRIM(W_SENTENCE,RIGHT=' ')
INVERT W_WORK
SET N = $INDEX(W_WORK,SEARCH= ' ')
SET W_LAST_WORD = $SUBSTR(W_WORK,LEN=(N - 1 ))
INVERT W_LAST_WORDSET W_SENTENCE = $SUBSTR(W_WORK,START=(N + 1))
INVERT W_SENTENCE
```

As a result of the example, W_LAST_WORD contains 'SENTENCE' and W_SENTENCE contains 'THIS IS A'.

LIST Statement

The LIST statement sends data to the RUNLIST output file. LIST is useful for displaying simple messages or for displaying the contents of fields for debugging. The Report Definition Facility (see Generating Reports) and the PRODUCE statement (described in this chapter) provide more flexible report specification.

The first format of the LIST statement sends variables or literals to the RUNLIST output file. In z/OS batch, this is the file with the RUNLIST DD name. In VSE batch, this is SYSLIST. This is the file in the output library with an output name identical to the main program. You can browse or print this output later.

This statement has the following format:

```
LIST [ERROR | list_specification]
```

list_specification

Specifies the data to list. The format is:

```
{numeric_field   }[     {numeric_field    }]
{date_field      }[     {date_field       }]
{alpha_expression}[ [,] {alpha_expression }]
{flag            }[     {flag             }] ...
{SKIP            }[     {SKIP             }]
{PANEL panelname }[     {PANEL panelname  }]
{NEWPAGE         }[     {NEWPAGE          }]
```

For more information about definitions of numeric_field, date_field, alpha_expression, and flag, see section, PDL Language Elements in the chapter one.

SKIP

Causes the listing to skip to a new line.

PANEL panelname

Writes an image of the named panel with the current field values. A panel wider than 132 characters is truncated.

NEWPAGE

Causes the listing to skip to the top of the next page.

ERROR (Only in an error procedure or WHEN ERROR clause.)

Lists information about an error condition that terminated a run. LIST ERROR automatically displays the value of the $RETURN-CODE function. For further information on error conditions, refer to the $ERROR and $RETURN-CODE functions in Chapter 5 and error procedure in this chapter.

On output, items on the same line are separated from each other by one blank.

If a numeric or date field is identified in a list_specification, the value of the field is first converted internally with $STRING and the result is listed.

For a variable length field, the actual length is listed.

An item with a value of null is listed as a question mark (?) or by the character specified using the SET REPORT NULLSYM command.

You can override the destination of the LIST output with an ASSIGN REPORT RUNLIST statement (or command). This statement lets you assign a destination and a disposition to the RUNLIST output. For more information, see the ASSIGN REPORT command in the *Command Reference Guide*.

Example

```
LIST 'THE ANSWER IS' X, Y 'ON', Z
:  This results in the concatenation of the values
:  of the designated literals and identifiers
:  with one blank between each.
```

If X='PLACE', Y='HAT', and Z='TABLE' the output line is:

```
THE ANSWER IS PLACE HAT ON TABLE
```

LOOP Statement

The LOOP statement executes one or more statements repeatedly under the control of one or more conditions. You can also perform looping implicitly with the FOR EACH statement. (See the description of the FOR EACH statement.)

This statement has the following format:

```
[<<label>>]
   LOOP

       statements

   [ {WHILE}           ]
   [ {UNTIL} condition ] ...
   [    statements     ]

   ENDLOOP

[<<label>>]
   LOOP numeric_expression_1 TIMES
    statements

   [{WHILE}           ]
   [{UNTIL} condition ] ...
   [      statements   ]

   ENDLOOP

[<<label>>]
   LOOP VARYING identifier
       [FROM numeric_expression_2]
       [BY numeric_expression_3]
     [[UP  ]                         ]
     [[DOWN] THRU numeric_expression_4]
                 statements
```

```
[ {WHILE} condition ] ...
[ {UNTIL}           ]
[     statements    ]
   ENDLOOP
```

<<label>>

An optional 1- to 15-character label for a LOOP construct. The label on the LOOP identifies the LOOP construct. You can use it to refer to the LOOP from other statements, such as the QUIT or PROCESS NEXT statements, or as the operand of certain functions, such as $COUNT.

WHILE

A condition that indicates that the loop executes repeatedly as long as the condition remains true. If the condition is false or unknown, the loop is terminated. You can use multiple WHILE clauses.

UNTIL

A condition that indicates that the loop executes repeatedly until the condition becomes true (as long as the condition remains false or unknown). You can use multiple UNTIL clauses.

numeric_expression_1 TIMES

Specifies the maximum number of times the loop executes. If the value of this expression is less than or equal to zero, no iterations are performed. If the TIMES clause results in a number that has decimal places, the number of iterations is rounded to the next higher integer.

VARYING clause

Specifies the identifier of a numeric field whose value is varied each time through the loop. There can be only one VARYING clause in a LOOP.

- **Identifier-**Specifies the identifier of the numeric field whose value is increased or decreased each time through the loop. The value of the field is varied by the value of numeric_expression_3. This field must be modifiable.
- **FROM clause-**Specifies an initial value from which the number of repetitions of the LOOP is counted.

- **numeric_expression_2-**A numeric expression that sets the initial value from which the number of repetitions is counted. The default is 1.
- **BY clause-**Specifies the value that increases or decreases the value of the identifier each time through the LOOP. The BY value must be positive when varying UP and negative when varying DOWN.
- **numeric_expression_3-**A numeric expression that specifies the value that increases or decreases the value of the identifier. The default is 1 when you specify UP; -1 when you specify DOWN.
- **DIRECTION clause-**Specifies the direction of incrementing through the loop and a value to compare with the value of the identifier. The direction clause has the following format:

  ```
  [UP  ]
  [DOWN] THRU numeric_expression_4
  ```

  The loop is terminated when using UP and the value of the identifier exceeds this value or when using DOWN and the value of the identifier falls below this value.
- **UP-**Specifies that the value of the identifier is increased by the value of numeric_expression_3. The LOOP terminates when the value of the identifier exceeds the value of numeric_expression_4. The value of numeric_expression_3 must be positive or the LOOP cannot terminate.
- **DOWN-**Specifies that the value of the identifier is decreased by the value of numeric_expression_3. The LOOP terminates when the value of the identifier falls below the value of numeric_expression_4. The value of numeric_expression_3 must be negative or the LOOP cannot terminate.
- **numeric_expression_4-**A numeric expression that sets the value that is compared to the value of the identifier.

ENDLOOP

A reserved word that designates the end of a LOOP construct.

The numeric expressions used as arguments in this statement are not nullable.

In PDL, WHILE and UNTIL indicate whether to continue or to quit if the condition is true. WHILE and UNTIL imply nothing about testing before or after each iteration of the loop. The location of the tests in the loop is determined by the placement of the WHILE and UNTIL statements, as shown in the examples.

Statements can appear before and after a WHILE or UNTIL clause. Placement of the statements in relation to the tests affects whether the statements ever executes.

When a PROCESS NEXT statement is encountered in a loop, the current loop terminates execution and the loop is reiterated.

When a QUIT statement is encountered in a loop, execution continues with the statement that follows ENDLOOP.

In the following loop, the test is made before any statements are processed. Therefore, the statements cannot execute at all. When using UNTIL, no iterations of the loop are performed if the test is true. When using WHILE, no iterations of the loop are performed if the test is false or unknown.

```
LOOP
UNTIL/WHILE condition
  statements
ENDLOOP
```

In the following loop, the test is first made after the statements were processed for the first time. Therefore, the statements in the following LOOP execute at least once.

```
LOOP
  statements
UNTIL/WHILE condition
ENDLOOP
```

In a loop of the following form, the test is first performed after the first set of statements processes and can exit the LOOP before the second set of statements is processed.

```
LOOP
  statements_1
UNTIL/WHILE condition
  statements_2
ENDLOOP
```

In the following loop, the VARYING clause processes array items. This loop varies LOOP-INDEX in descending order, beginning the process at 10 and continuing until 1 process. After the loop ends, LOOP-INDEX has a value of zero.

```
LOOP
VARYING LOOP-INDEX FROM 10 BY -1 DOWN THRU 1
    CALL CHECK USING A (LOOP-INDEX)
ENDLOOP
```

You can use FROM, BY, and THRU clauses in any order.

The following is an infinite loop.

```
LOOP
VARYING LOOP-INDEX FROM 100 BY 1 DOWN THRU 100
    statements
ENDLOOP
```

This loop repeatedly transmits a panel until you enter TRANSCODE T on the panel or press the PF3 key. If the TRANSMIT does not present the application with terminating data, TRANSCODE determines further processing on each subsequent LOOP iteration.

```
<<MAIN>> PROCEDURE
  LOOP
   TRANSMIT MAINPNL
  UNTIL $PF3
  UNTIL TRANSCODE = 'T'
  SET MAINPNL.MSG = ' '
  SELECT TRANSCODE
  WHEN 'A'
   DO ADD_REC
  WHEN 'B'
   DO DEL_REC
  WHEN OTHER
   DO OTHER_PROC
  ENDSEL
 ENDLOOP
ENDPROC
```

This loop processes a sequential file until the first header record (in a group of records with multiple types) is encountered. If currently positioned at a non-header, records are read until a header is found.

```
<<POSITION-HDR>>
  LOOP
     FOR NEXT MASTER-FILE
        : BYPASS
     WHEN NONE
         QUIT POSITION-HDR
     ENDFOR
  UNTIL MASTER-FILE.RECORD-TYPE = 'A'
  ENDLOOP
  DO PROCESS-A
```

Examples

The following are further examples that show various positions of the loop termination test and expressions used for the VARYING clause.

```
LOOP
UNTIL BALANCE NOT > AMOUNT
   DO PRINT-BALANCE
   SUBTRACT AMOUNT FROM BALANCE
ENDLOOP
LOOP VARYING I FROM 1 BY 1 THRU N
   MOVE STATE(I) TO X-STATE
WHILE NOT ERROR-COND
   DO PROC-STATE
ENDLOOP
LOOP
   statements
WHILE condition-1
   statements
WHILE condition-2
   statements
ENDLOOP
LOOP VARYING I
      FROM X + 3
      BY 2
      THRU (A + B)/2
      statements
ENDLOOP
```

MOVE Statement

The MOVE statement transfers data from a source to a target. The MOVE statement does not modify the value of the source and the original data remains in the source after it is moved to the target.

The statement MOVE TO Numeric Field has the following format:

```
MOVE {alphanumeric_expression| numeric_expression | NULL}  TO numeric_field ...
```

The statement MOVE TO Alpha Field has the following format:

```
MOVE {alphanumeric_expression| numeric_field | numeric_literal | group | NULL}  TO
alphanumeric_field ...
```

The statement MOVE TO Group has the following format:

```
MOVE group1 TO group2{BY NAME| POSITION} [USING {$EDIT| $STRING } [RULES]]
```

MOVE (MOVE TO Numeric Field)

Moves a value to an elementary numeric field according to the following rules:

If both the source and target are numeric fields, the value is moved by alignment of an implicit decimal point, with truncation of low-order decimal digits (when necessary). If high order significant digits are lost, an overflow error condition is raised and the error procedure is executed (see the Error Procedure topic in this chapter.

If the target is a date field, then the source cannot be an alphanumeric

When a value from an alphanumeric source is moved to a non-date numeric target, the value of the alphanumeric source is first converted to numeric by applying the $NUMBER function to the alphanumeric item (see the $NUMBER function). If the source contains non-numeric characters, an execution-time error occurs.

If the source is the keyword NULL or evaluates to the null value, the target field must be nullable. It is set to the null value.

MOVE (MOVE TO Alpha Field)

Moves a value to an elementary alphanumeric field according to the following rules:

- If both the source and target are alphanumeric fields, the value is moved as follows:
- If both source and target are the same length, an exact copy is made.
- If the length of the value of the source is longer than the target, truncation occurs on the right.
- If the target is longer than the length of the value of the source, then the result is padded on the right with blanks.
- If the target is a variable length field, then the length of the target becomes the length of the source, up to the maximum length defined for the variable length field.
- If the source is a variable length field, then its length is the length of the current value. This length is compared with the target as described above and is padded or truncated.
- If the source is a non-alpha group, it is treated as an alphanumeric field whose length is the same as the size of the group. However, subordinate numeric fields are not converted. The hexadecimal values are simply moved.

You cannot specify restricted groups. For more information, see the Restrictions on Non-Alpha Groups topic in the "Procedure Definition Language Concepts and Language Elements" chapter.

PDF can convert low values moved to an alphanumeric field in a panel with the $LOW function (or by some other means) to special characters. For a description of the Input Fill Character, see the *Creating Panel Definitions Guide*.

When low values are moved to an alphanumeric panel field, CA Ideal considers the field EMPTY. The next TRANSMIT fills this EMPTY field with the Output Fill Character when it is next displayed.

When a value from a non-date numeric source is moved to an alphanumeric target, the data in the numeric source is first converted to alphanumeric by applying the $STRING function (see the section on the $STRING function in "Symbolic Debugger commands" chapter). If you do not want to use the $STRING rules, you can specify a $EDIT function in a numeric expression as the source. Use the default PIC or specify one of the other edit patterns. See the section on $EDIT in the "Symbolic Debugger commands" chapter).

If the conversion results in a value that is longer than the target field after any leading blanks were removed, digits are truncated from the right to fit. For example, if NUM is a three-digit numeric field and STR is a two-character alphanumeric field, the expression results in STR having a value of '12' if NUM has a value of 123. If, however, NUM has a value of 12 or 1, STR has a value of '12' or '1' respectively.

```
MOVE NUM TO STR
```

You cannot move an arithmetic expression or numeric function to an alphanumeric target without first moving it to a numeric target.

You cannot move a value directly from a date type source to an alphanumeric target. Use the $DATE function as an alphanumeric expression to convert the date field as part of the MOVE.

If the source is the keyword NULL or evaluates to the null value, the target field must be nullable. It is set to the null value. If a null value expression is moved to a target that is not nullable, it causes a runtime error.

MOVE (MOVE TO Group)

- Moves data from one group to another. The following rules apply:
- Both the source and the target must be groups.
- Moving a value from each field in the source group to a field in the target group is subject to the rules for moving values described in the Move TO Alpha and Move TO Group formats.
- When values are moved from a non-alpha group to an alpha target, the default is to convert numeric data to alphanumeric by applying a STRING function. You can specify that a $EDIT function to use instead. See the following options USING $EDIT RULES and USING $STRING RULES.

BY NAME

Moves the value from each field in the source group to an identically named field in the target group, if one exists. OCCURs in the respective groups (if any) must be compatible. Only values from fields that have the same names are subordinate to the respective groups and are elementary are moved. Redefinitions in the sending group are not eligible sources; however, redefinitions in the receiving group are eligible targets. A compile-time error message is issued if the number of occurrences does not match. A runtime error occurs if the number of occurs depending on or parameter field occurrences does not match.

For example, if the source group is

```
Level          Field              Occur
1              A
  2             B                 2
  2             C                 2
```

and the target group is,

```
Level            Field              Occur
1                X
  2              Y                   2
    3              B
    3              C
```

then the statement,

```
    MOVE A TO X BY NAME
```

moves the values of the Bs and Cs in the source group to fields with corresponding names in the target group.

BY POSITION

Moves the value of the first elementary field in the source group item to the first elementary field in the target group item, regardless of its name; the second to the second, and so on. The group structures must be compatible as follows:

Structures must have the same number of elementary fields.

OCCURs values must be identical and at the same relative level.

Test structures for compatibility with the following steps:

1. Remove any OCCURs values from the high level group entries in both the source and target.
2. Remove any fields or groups with REDEFs in both the source and target. If a group is removed, all its subordinate fields are also removed.
3. Remove each subordinate group entry except any that have OCCURs.
4. Renumber the levels for the fields that are left.

Note: No field values are moved unless the structures that remain after this test is applied are compatible.

USING $EDIT RULES

In a group move only, specifies to convert numeric fields to alphanumeric by applying a $EDIT function with an edit pattern of PIC 9(n), where n is the number of integer places in the field. The default uses a $STRING conversion.

USING $STRING RULES (Default)

In a group move only, performs numeric conversion using the $STRING rules.

Examples

```
MOVE QUANTITY_ORDERED * UNIT_PRICE TO CHARGE

MOVE CURRENT_AMOUNT TO AMOUNT_RQRD
MOVE $NUMBER (PAY_AMT) TO PAY_AMT_NUMERIC
MOVE ((Y + Z)*W)/(A - B) TO Z
MOVE A + ($REMAINDER(M,N) - $ROUND(J,1))/3.6 TO L
MOVE $STRING (X,Y,Z) TO ALPHA
```

Examples: MOVE BY NAME

Structures:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B				3	
	3	C	X		2	4	
	3	D	X		2		
	3	E	X		2		

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	X					
	2	C	X		2	4	
	2	D	X		2		

Statement a:

```
MOVE B(1) TO X BY NAME
```

Result a:

The values of all occurrences of C(1,n) and D(1) in structure 1 are moved to C and D in structure 2 (B is a group).

Statement b:

```
MOVE A TO X BY NAME
```

Result b:

Invalid MOVE because structures 1 and 2 do not match. The number of subscripts for C in structure 1 is 2, and the number for C in structure 2 is 1.

Example

Structures:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B				3	
	3	C	X		2	5	
	3	D	X		2		
	3	E	X		2		

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	X					
	3	C	X		2	4	
	3	D	X		2		

Statement:

```
MOVE B(1) TO X BY NAME
```

Result:

Invalid MOVE because OCCUR of 5 for C in structure 1 and OCCUR of 4 for C in structure 2 cause the structures not to match.

Example

Structures:

```
    Level  Field   Type  Int   Chars   Occur   Value/Comments
1)    1      A
      2      B                           3
      3      C      X           1        4
      3      D      X           1
      3      E      X           1

    Level  Field   Type  Int   Chars   Occur   Value/Comments
2)   1      AA
     2      BB      X           1
     3      E       X           1       3
     4      C                           4
     2      D       X                   3
```

Statement a:

```
MOVE A TO BB BY NAME
```

Result a:

Moves only the values of all occurrences of C because it is the only elementary field that has the same name in both structures. Although E is part of BB, it is not moved because it is a group.

Statement b:

```
MOVE A TO AA BY NAME
```

Result b:

Moves the values of all occurrences of C and D because both C and D are the only elementary fields that have the same names in both structures.

Examples: MOVE BY POSITION

This first example of MOVE BY POSITION illustrates the rules by which data structures are tested for compatibility using the rules outlined in the description of MOVE BY POSITION. Subsequent examples are simpler and their explanations assume the reader understands the rules illustrated in this first detailed example.

Structures:

```
    Level  Field   Type  Int   Chars   Occur   Value/Comments
1)  2      A                            2      REDEFINED Y
    3      B      X           5         5
    3      C      X           5
    3      D                                   REDEFINES C
    4      E      X           2
    4      F      X           3
    3      G      N           2

    Level  Field   Type  Int   Chars   Occur   Value/Comments
2)    2      M                          2      REDEFINES Q
      3      N
      4      O      X           5       5
      4      P      X           4              REDEFINES P
      4      Q      X           4
      4      R      X           1
```

Statement:

```
MOVE A TO M BY POSITION
```

Result:

This is a valid move. The structures are tested for compatibility with the following steps:

1. Removes any OCCURs values from group entries in both the source and target. After applying 1., the OCCUR values 2 in structure 1 and 2 in structure 2 are removed.
2. Removes any fields or groups with REDEFs in both the source and target. If a group is removed, all its subordinate fields are also removed. After applying 2., the fields D, E and F in structure 1 and Q in structure 2 are removed.
3. Removes each subordinate group entry except any that has OCCURs. After applying 3., the field N in structure 2 is removed.

4. Finally, the levels are renumbered for the remaining fields. After applying 4., the result is:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B	X		5	5	
	2	C	X		5		
	2	G	N		2		

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	M					
	2	O	X		5	5	
	2	P	X		4		
	2	R	X		1		

The move is now formed as follows: The values of all occurrences of B are moved to O, the value of C is moved to P, the value of G is moved to R.

If field O had an OCCURs of 4 instead of 5 after the test for compatibility, the structures would not match and the MOVE statement would be invalid.

Example

Structures:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B	N		2		
	2	C	X		3		
	2	D	X		3		

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	E					
	2	F	N		2		
	2	G					
	3	H	X		3		
	3	I	X		3		

Statement:

```
MOVE A TO E BY POSITION
```

Result:

Valid MOVE BY POSITION: The value of B moves to F, the value of C moves to H, the value of D moves to I (G is a group).

Example
(Invalid Move)

Structures:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B	N		3		
	2	C	X		2	2	
	2	D	X		2	2	

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	E					
	2	F	N		3		
	2	G				2	
	3	H	X		2		
	3	I	X		2		

Statement:

```
MOVE A TO E BY POSITION
```

Result:

Invalid move because of different structure. Although there are the same number of elementary fields, the depth of the structure in E is 3 and that of A is only 2. Altering structure E according to the rules does not alter the structure since G (a lower-level group) has an OCCURs clause.

Example

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	A					
	2	B	X		3		
	2	C	X		3		
2)	1	D					
	2	E			2		
	3	F	X		3		
	3	G	X		3		

Statement:

```
MOVE A TO E(1) BY POSITION
```

Result:

Valid MOVE statement: The value of B moves to F(1) and the value of C moves to G(1). Even though E is a repeating group, the subscript indicates that only the value of the first occurrence of the group participates in the move. This, in effect, makes the two structures compatible.

Non-Alpha to Alpha Group Move

The following examples show group moves of non-alpha groups to alphanumeric structures:

Structures:

	Level	Field	Type	Int	Chars	Occur	Value/Comments
1)	1	NUM_DATE					
	2	NUM_YY	U	Z	2		86
	2	NUM_MM	U	Z	2		02
	2	NUM_DD	U	Z	2		09
	1	NUM_SSN					
	2	SSN1	U	Z	3		36
	2	SSN2	U	Z	2		8
	2	SSN3	U	Z	4		22

	Level	Field	Type	Int	Chars	Occur	Value/Comments
2)	1	GROUP_DATE					
	2	ALPHA_YY	X		2		
	2	ALPHA_MM	X		2		
	2	ALPHA_DD	X		2		
	1	ALPHA_DAT	X		6		
	1	WORK_SSN					
	2	SSN1	X		3		
	2	SSN2	X		2		
	2	SSN3	X		4		
	1	ALPHA_SSN	X		11		

Statements:

```
MOVE NUM_DATE TO GROUP_DATE BY POSITION USING $EDIT RULES
       :Result is 860209
MOVE NUM_DATE TO GROUP_DATE BY POSITION USING $STRING
       :Result is 86 2 9
MOVE NUM_DATE TO ALPHA_DAT
       :Result is 860209
MOVE NUM_SSN TO WORK_SSN BY NAME USING $EDIT
       :Result is 036080022
MOVE $EDIT(WORK_SSN,PIC='XXX-XX-XXXX') TO ALPHA_SSN
       :Result is 036-08-0022
```

Note: The numeric fields in the groups NUM_DATE and NUM_SSN are defined as unsigned, zoned decimals so that, when their hexadecimal values are copied into the corresponding alpha fields, the results represent the original values. This is not the case if the fields are signed numerics or are in binary or packed internal format.

NOTIFY Statement

The NOTIFY statement transmits data or a message to the message line in an online or batch environment. You can also send the message to the operator console. NOTIFY is useful for displaying a message with information about errors, instructions to continue, warnings, and so on, to the user's or operator's terminal.

This statement has the following format:

```
NOTIFY list_specification [TO CONSOLE]
```

list_specification

Specifies the data to transmit. The format is as follows:

```
{flag                   }[    flag                   ]
{numeric_field          }[    numeric_field          ]
{date_field             }[[,] date_field             ] ...
{alphanumeric_expression}[    alphanumeric_expression ]
```

For more information about flag fields, numeric fields, date fields, and alphanumeric expressions, see the PDL Language concepts topic in the "Procedure Definition Language Concepts and Language Elements" chapter.

TO CONSOLE

The CONSOLE clause lets the application send a message to the operator console. The message is also sent to the z/OS JESLOG or VSE POWER LOG in batch or to the CICS System Message Block online.

The NOTIFY message is sent from the program to the message line when the next TRANSMIT statement executes. If there are multiple NOTIFY statements, only the last NOTIFY message before the TRANSMIT is sent.

The NOTIFY message is cleared from the message line when the next statement in the program executes after the TRANSMIT. If no TRANSMIT occurs between NOTIFY and the end of a run, the message appears on the next CA Ideal system panel. In this instance, the NOTIFY message overrides the RUN COMPLETED message.

If the TO CONSOLE clause is used, the message is sent to the operator's console when the NOTIFY statement is executed.

The maximum length of a NOTIFY message is 79 characters. The maximum length of a NOTIFY message sent to a console is 72 characters.

A nullable data item with a value of NULL is shown as a question mark (?) or the character specified using the SET REPORT NULLSYM command.

Example

The following example transmits a user panel (USRPANEL) in a loop and performs a <<CHECK_ERRORS>> procedure that verifies field information.

The PNL_SSN field is checked to allow only numeric characters. If an error is found, the ERRORS field is updated and the NOTIFY statement holds this message to send with the next TRANSMIT statement.

The EMP_NUM field is checked to allow only numeric characters. If an error is found, the ERRORS field is updated and the NOTIFY statement holds this message to send with the next TRANSMIT statement.

Only the last NOTIFY message is sent for each TRANSMIT. In this example, NOTIFY eliminates the need for a separate message field in USRPANEL.

```
<<NOTIFYEX>> PROC
      SET ERRORS EQ 1
      LOOP UNTIL ERRORS = 0
          TRANSMIT USRPANEL
          DO CHECK_ERRORS
      ENDLOOP
ENDPROC

<<CHECK_ERRORS>> PROC
      SELECT FIRST ACTION
      WHEN NOT $VERIFY(PNL_SSN, AGAINST = NUMERIC)
          SET ERRORS = ERRORS + 1
          NOTIFY $STRING(PNL_SSN, ' INVALID SSN')
      WHEN  NOT $VERIFY(PNL_EMP_NUM, AGAINST = NUMERIC)
          SET ERRORS = ERRORS + 1
          NOTIFY $STRING(PNL_EMP_NUM, ' INVALID EMP_NUM')
      WHEN OTHER
          SET ERRORS = 0
      ENDSELECT
ENDPROC
```

Procedure

A procedure is a named, functional collection of statements. Procedures can divide a program or subprogram into logical subcomponents.

This statement has the following format:

```
<<procedure_name>> {PROC      }
                   {PROCEDURE }
         statements
[ENDPROC     ]
[ENDPROCEDURE]
```

procedure_name

A 1- to 15-character user-defined label for the procedure. The chevrons (<< and >>) are required. The chevrons are not included in the label length. Blanks are left- and right-justified. Internal blanks are not allowed.

PROC|PROCEDURE

A reserved word that designates the beginning of a new procedure. The procedure name and this reserved word are optional for the first or main procedure of the program.

ENDPROC|ENDPROCEDURE

A reserved word that designates the end of the procedure. This reserved word is optional and, when omitted, the next procedure or the end of the program terminates the procedure.

You can omit the label and the reserved word PROC or PROCEDURE on the first (or main) procedure only.

Any statements occurring after an ENDPROCEDURE and before the next PROCEDURE are treated as errors.

You can use a procedure with the name <<ERROR>> in a program to handle execution time errors at runtime without necessarily aborting the RUN or take action before quitting execution of the program. See the Error Procedure topic in this chapter.

Example

This example illustrates two procedures, which are the main procedure and procedure ADD_REC. Two other procedures referenced by the main procedure, DEL_REC and OTHER_PROC, are not shown.

```
<<MAIN>> PROCEDURE
  LOOP
   TRANSMIT MAINPNL
  UNTIL TRANSCODE = 'T'
    SET MAINPNL.MSG = ' '
    SELECT TRANS_CODE
    WHEN 'A'
        DO ADD_REC
   WHEN 'B'
        DO DEL_REC
   WHEN OTHER
        DO OTHER_PROC
   ENDSEL
  ENDLOOP
ENDPROC
```

```
<<ADD_REC>> PROCEDURE
    TRANSMIT ADDPNL CLEAR
    FOR NEW EMPLOYEE
        SET EMPLOYEE = ADDPNL BY NAME
        SET MAINPNL.MSG = 'EMPLOYEE ADDED'
    WHEN DUPLICATE
        SET MAINPNL.MSG = 'RECORD ALREADY ON FILE'
    ENDFOR
ENDPROC
```

PROCESS NEXT Statement

The PROCESS NEXT statement terminates the current iteration and initiates the next iteration of a repetitive construct (LOOP, FOR EACH/FIRST *n*/ANY). If the current construct is a FOR, any data record acquired for update is released and no updates take place. The process continues with the next record, if any.

This statement has the following format:

```
PROCESS NEXT [label]
```

PROCESS NEXT

Without a label, PROCESS NEXT must appear only in the lexical scope of a LOOP or FOR EACH construct. With a label, PROCESS NEXT must appear in the logical scope of a LOOP or FOR EACH construct.

When a PROCESS NEXT statement appears in the scope of a FOR EACH, any updates to the current dataview record are abandoned, even for fields whose values already were changed.

label

The label of the LOOP or FOR construct for which the current iteration is terminated. The PROCESS NEXT label statement must be in the logical scope of the construct identified by label, that is, PROCESS NEXT must reference the current construct or one at a higher logical level.

When constructs are nested, for example, A invokes B (with a DO B statement), B invokes C, C invokes D, and so on, all procedures in the series of invocations down to the most recently invoked procedure are active. In the above series of invocations, a QUIT B issued from B, C, or D makes B, C, and D inactive and returns control to A. This also applies to nested LOOP or FOR.

The PROCESS NEXT statement, without a label, must appear physically in the program text between FOR/ENDFOR or LOOP/ENDLOOP construct where the statement applies. The FOR or LOOP construct so referenced need not have an explicit label. PROCESS NEXT, without a label, should never appear in a WHEN NONE clause of a FOR construct.

You can always code the PROCESS NEXT statement with a label, regardless of where it appears.

When the PROCESS NEXT statement is executed, the construct where its label refers must be undergoing active iteration. If more than one iterative process can execute a common procedure, take care when specifying PROCESS NEXT to ensure that only the iterative process currently executing is processed.

When PROCESS NEXT executes, the ENDLOOP or ENDFOR statements are not executed.

A PROCESS NEXT in the scope of a FOR construct abandons any modifications to the file caused by the FOR construct, since no modifications are applied until the ENDFOR, with one exception: A DELETE is performed immediately and is not aborted by a PROCESS NEXT.

Example

```
<<EMP>>
    FOR EACH EMPLOYEE
      WHERE DEPT = 'D' AND JOB_CODE = 'J'
        DO NOTE_DJ_EMP
      <<DEP>>
          FOR EACH DEPENDENT
             DO NOTE_DEP
             IF DEP_AGE > 21
              DO TOO_OLD
              PROCESS NEXT DEP
             ENDIF
             DO ANAL_DEP
         ENDFOR
      IF FOUND_ENOUGH_EMP       QUIT EMP
      ENDIF
ENDFOR
```

Example

```
<<MAIN>> PROCEDURE
    SET EMPLOYEE_CO = 0
    <<EMP>>
        FOR EACH EMPLOYEE
          SET FOURTH_QTR_SALES = SALES (10)
            + SALES (11) + SALES (12)
          IF FOURTH_QTR_SALES < 1000
            PROCESS NEXT EMP
          ENDIF
          MOVE 0 TO TOTAL_SALES
          MOVE 1 TO LOW_SUB
          MOVE 1 TO HIGH_SUB
          LOOP VARYING SEARCH_SUB FROM 1 BY 1 THRU 12
            ADD SALES (SEARCH_SUB) TO TOTAL_SALES
            IF SALES (LOW_SUB) > SALES (SEARCH_SUB)
              MOVE SEARCH_SUB TO LOW_SUB
            ENDIF
            IF SALES (HIGH_SUB) < SALES (SEARCH_SUB)
              MOVE SEARCH_SUB TO HIGH_SUB
            ENDIF
          ENDLOOP
          LIST EMPLOYEE.NAME  SALES (LOW_SUB)
            SALES (HIGH_SUB)  TOTAL_SALES
          ADD 1 TO EMPLOYEE_COUNT
          IF EMPLOYEE_COUNT > 99 :illustration only,
             QUIT EMP            :could have been done
          ENDIF                 :with
       ENDFOR                   :FOR FIRST 99 EMPLOYEE
                                :at beginning
                                 :of EMP PROCEDURE
ENDPROC
```

PRODUCE Statement

The PRODUCE statement generates a report that must be previously defined with the Report Definition Facility. The PRODUCE statement usually is contained in a FOR or LOOP structure. Each execution of the PRODUCE statement generates one detail group comprising one or more physical lines. (For more information, see the *Generating Reports Guide*.) The PRODUCE command (see the *Command Reference Guide*) generates a report facsimile.

This statement has the following format:

```
PRODUCE report_name[group_name]
```

report_name

The one- to eight-character name of the report definition for which output is generated.

group_name

The three- to eight-character name of a group in the detail section of the report. If omitted, the primary group is assumed.

Note: Producing a secondary group before its primary group can cause unpredictable results.

Page breaks, control breaks, headings, summaries, and so on, are produced automatically according to the report specification (see the *Generating Reports Guide*).

In batch, each eight-character report-name corresponds to the name of a DD statement and, therefore, must be unique in the run. If the report-name contains hyphens (-) or underscores (_), you must use an ASSIGN statement to provide a legal DD name to the operating system.

All fields in the detail group must have values at the time the PRODUCE statement is issued or a runtime error is produced.

A PRODUCE statement can reference only reports that were specified in the program's resource table.

A PRODUCE statement activates a report and it remains active until the application terminates or until the program or the report is released. For more information, see the RELEASE statement topic in this chapter.

A maximum of 15 reports can be active simultaneously.

Example

```
FOR EACH EMPLOYEE
   WHERE STATE_ADDRESS = 'TX' AND CITY_ADDRESS = 'DALLAS'
   PRODUCE EMPRPT
ENDFOR
```

QUIT Statement

The QUIT statement causes the flow of control to abandon one or more current constructs, procedures, or the current program. Subsequent flow of control continues as if the affected constructs, procedures, or program were exited normally.

This statement has the following format:

```
     [label-of-FOR-or-LOOP                              ]
     [PROGRAM                                           ]
     [RUN                                               ]
QUIT [label-of-procedure                                ]
     [PROCEDURE                                         ]
```

QUIT

With no operands, terminates the current FOR or LOOP construct or procedure. The statement following the ENDFOR, ENDLOOP, or ENDPROC executes next. When a QUIT statement applies to a FOR construct, no updates to the current dataview record are applied.

label-of-FOR-or-LOOP

Terminates the label of the construct. The QUIT label-of-FOR-or-LOOP statement can only execute in the logical scope of that labeled construct or in an ERROR PROCEDURE construct.

PROGRAM

A reserved word that terminates the current program or subprogram. QUIT PROGRAM is not required at the normal end of a program since the end of the main procedure of a program implies a QUIT PROGRAM. The abbreviation PGM cannot be used in this statement.

RUN

A reserved word that terminates the current program and all other programs currently active in the run-unit.

label-of-procedure

The label of a procedure to terminate. This must be the current procedure or an active procedure that directly or indirectly invoked the current procedure.

PROCEDURE

A reserved word that terminates the current procedure. Control returns to the invoking procedure. If a QUIT is issued for the main procedure, QUIT PROCEDURE is the equivalent of QUIT PROGRAM.

Example

```
<<UP_DATE>> PROCEDURE
    <<FOR_1>>
         FOR NEW updatable_dvw_name
           MOVE 'X' TO dvw_field
           DO DETERMINE_CONT
           IF condition
              DO EXIT
           ENDIF
           DO REST_UPDATE
        ENDFOR
ENDPROC

<<EXIT>> PROCEDURE
      QUIT UP_DATE
ENDPROC
```

REFRESH Statement

The REFRESH statement resets all fields in the named panel to their initial values and attributes as defined in the panel definition. The REFRESH statement ensures that a fresh copy of the panel is available.

The *Creating Panel Definitions Guide* describes how to define and maintain screen panels. Several statements and built-in functions provide symbolic high-level panel processing (see the TRANSMIT, SET ATTRIBUTE and RESET statements).

This statement has the following format:

```
REFRESH panel_name
```

panel_name

The one- to eight-character name of the previously defined panel. You must define the panel-name in the RESOURCE section of the program where the statement appears.

Fields with an initial value set in LAYOUT are refreshed with that value, whether their initial attribute is protected or unprotected.

Fields that do not have an initial value set in LAYOUT (for example, spaces between start-field and end-field) are initialized to input-fill characters. When TRANSMIT occurs, fields still set to input-fill are reset with output-fill.

REFRESH often restores a panel for a new set of data after an old set of data completes processing. You can perform restoration of initial values and attributes here in a single operation rather than many RESET and SET ATTRIBUTE statements.

If you only need to reinitialize a few fields, use RESET on individual fields with SET ATTRIBUTE as an alternative to REFRESH. The REFRESH statement refreshes the panel object in memory with the panel object from the file; RESET and SET ATTRIBUTE change the panel object in memory.

Repeated use of REFRESH can increase overhead if performed unnecessarily.

RELEASE Statement

The RELEASE statement releases a panel, report, or subprogram after it is no longer needed as a resource of a program. You can use the RELEASE PROGRAM statement with a CA Ideal subprogram or a non-ideal subprogram run in batch that is defined not to load a new copy on each call.

If a subprogram is not explicitly released, it remains on call, ready to be called again until the run is completed. In a batch run or in an online run that does not cross transaction boundaries, the subprogram is held on call in virtual memory. In an online run that does cross transaction boundaries, unreleased programs are swapped out to auxiliary storage on TRANSMIT statements. In either case, releasing programs that are no longer needed frees machine resources, especially with batch applications or online runs that do not cross transaction boundaries.

However, releasing programs that one or more programs in an application frequently use can add runtime overhead by forcing them to be continually reloaded. For more information about RELEASE PROGRAM, see the *CA Ideal Administration Guide.*

This statement has the following format:

```
        {PANEL panel_name              }
RELEASE {REPORT rpt-name [[WITH] ABORT }
        {PROGRAM [subprogram]          }
```

panel_name

The name of the panel to release. You cannot use the abbreviation PNL in this statement.

rpt_name

The name of the report to release or the word RUNLIST. You cannot use the abbreviation RPT in this statement.

WITH ABORT

Prints the data specifications on the lines following the <<ABORT>> group heading on the Heading fill-in before releasing the report. This is useful for printing messages when a report is terminated abnormally.

subprogram

The name of the subprogram to release. You cannot use the abbreviation PGM in this statement. If you omit the name, the RELEASE applies to the current program.

A maximum of 64 panels can be active simultaneously in all programs and subprograms in a run. In an application that uses large numbers of panels, releasing panels that are no longer needed makes it less likely that this limit is reached. It also reduces the amount of temporary storage each session that uses the panel needs.

There is a limit of 15 reports that can be simultaneously active. When a report of one of many used in an application, it should be released when no longer needed to minimize the likelihood that the limit on simultaneously active reports is reached during the run. (For more information, see the Generating Reports Guide.)

If a released panel is referenced again (such as with a RESET or a TRANSMIT statement), a fresh copy is reloaded from the library. If a panel is not released and the program where the panel is a resource is not released, the panel maintains its values between calls to the program.

When a program that was released is called again, a new copy is loaded with a fresh copy of working data.

All unreleased panels and reports used in a run are released at the end of the run or when the program that transmitted or produced them is released. All unreleased programs used in a run are released at the end of the run.

If a panel is a resource in more than one program of a run unit, the panel is counted separately against the maximum. For this reason, transmit a panel be in only one subprogram of an application.

When a report is released, final control footings and summary lines are produced. If RELEASE REPORT is issued and no data was printed, the data specifications on the lines following the <<EMPTY>> group heading are printed when the report is released.

You can release a report for one of three reasons:

- Because a program finished with it and it is no longer needed as a program resource.
- So that a new report using the same report definition can be produced later in the same program. If the same report were reinvoked with a PRODUCE statement without first being released, any new lines produced for that report are added to the existing report rather than generated as a separate report.
- When a report is routed to a network printer. The RELEASE REPORT statement closes the report and schedules it for printing. The printing process begins when the printer becomes available.

The RELEASE REPORT RUNLIST statement releases the output of a LIST statement. This keeps the outputs of a series of LIST statements separate rather than accumulating them as a single RUNLIST output.

You can only use the RELEASE PROGRAM statement for CA Ideal subprograms that do not load a new copy each time they are called in batch.

You can issue the RELEASE PROGRAM statement from either the calling program or the called program. When issued from the called program without specifying a program name, the statement implies a QUIT PROGRAM and the RELEASE. This statement does not affect the logic of the calling program, only the performance.

You cannot RELEASE a program that directly or indirectly called the current program.

The released subprogram must be in the resource table of the releasing program or must be the releasing program. All resources of the specified subprogram are released, except the subprograms that are themselves resources of this subprogram.

RESET Statement

The RESET statement initializes a panel field or an entire panel to the input fill character. Other terminal attributes, such as panel color, protection, and highlight attributes are unchanged.

The *Creating Panel Definitions Guide* describes how screen panels are defined and maintained. Several statements and built-in functions provide symbolic high-level panel processing (see the REFRESH statement section and the TRANSMIT statement section in this chapter).

This statement has the following format:

```
         [panel_field]
RESET    [panel_name ]
```

panel_field

Specifies the identifier of a panel field to reset.

panel_name

Specifies the one- to eight-character name of a previously defined panel.

A single field (either protected or unprotected) is reset when a panel field identifier is specified.

All unprotected fields (and only unprotected fields) in a panel are reset if a panel name is specified.

CA Ideal transforms any field that was initialized to the input fill character with a RESET statement and was not modified by the program before TRANSMIT into output-fill characters before presentation as panel-output.

SELECT Statement

The SELECT statement executes one or more of several courses of action based on one or more conditions. The SELECT statement has three formats.

This statement has the following format:

The Format 1 of the SELECT statement selects the set of statements that follows the first WHEN value that matches the value of the select-subject. Only the set of statements that follows the first matching value executes.

```
SELECT select_subject
         {numeric_expression } [   {numeric_expression} ]
    WHEN {alpha_expression   } [OR {alpha_expression  } ]...
         {NULL               } [   {NULL              ] ]

      statements

         {numeric_expression } [   {numeric_expression} ]
    WHEN {alpha_expression   } [OR {alpha_expression  } ]
 ...     {NULL               } [   {NULL              } ]

      statements
    .
    .
    .
   [     {NONE  } ]
   [WHEN {OTHER } ]
   [   statements ]

   [WHEN ANY     ]
   [   statements]

 {ENDSEL     }
 {ENDSELECT }
```

select_subject

The identifier of a numeric or alphanumeric expression whose value determines the action selected.

WHEN clause

Specifies a possible value (or values). If the value specified in the WHEN clause matches the value of select_subject, the statements that follow the WHEN clause execute. In Format 1, only the statements that follow the first WHEN condition to test true execute, optionally followed by a WHEN ANY clause.

numeric_expression|alpha_expression|NULL

The value that compares select_subject. You can combine expressions, including NULL, using OR.

WHEN OTHER|NONE

An optional postscript that specifies that when none of the values listed matches the value of select_subject, the statements following the WHEN OTHER or WHEN NONE execute. The reserved words OTHER and NONE are interchangeable.

WHEN ANY

An optional postscript that specifies that when any WHEN value matches the value of select_subject, the statements that follow the WHEN ANY execute in addition to the statements that follow the equal case.

ENDSEL|ENDSELECT

Reserved words that terminate the SELECT construct. If SELECT constructs are nested, the most recent unterminated SELECT construct is terminated by the first occurrence of ENDSEL or ENDSELECT. Each SELECT in a nested SELECT construct must have a corresponding ENDSEL or ENDSELECT.

Note: A select_subject that evaluates to the null value matches NULL in a WHEN clause. It does not match an expression in a WHEN clause that also evaluates to the null value.

Example

```
SELECT TRANS_CODE
WHEN 'A'
   DO ADD_RECORD_PROC
WHEN 'D'
   DO DEL_RECORD_PROC
WHEN 'P'
   DO PURCHASE_PROC
WHEN 'R'
   DO RECEIPT_PROC
WHEN ANY
   DO LOG_TRANS
WHEN OTHER
   DO INVALID_CODE
ENDSEL
```

Format 2 of the SELECT statement selects the statements that follow the first true condition in a series of conditions. Only the statements that follow the first true condition execute.

```
SELECT [FIRST [ACTION]]
   WHEN condition
      statements

       [WHEN condition]
       [statements    ] ...

   [     {NONE  }]
   [WHEN {OTHER }]
[  statements ]

   [WHEN ANY     ]
[  statements ]

   {ENDSEL    }
   {ENDSELECT }
```

FIRST [ACTION]

Optional reserved words that you can add for readability.

WHEN condition

Specifies a condition. (For an explanation of valid conditions, see the "Procedure Definition Language Concepts and Language Elements" chapter.) If this is the first true condition in the SELECT, only the statements that follow it execute, followed by a WHEN ANY clause if one is specified. If more than one condition is true, only the statements associated with the first condition execute.

WHEN OTHER/NONE

An optional postscript that specifies that, when none of the previously specified WHEN conditions listed is true, the statements that follow the WHEN OTHER or WHEN NONE execute. You can use the reserved words OTHER and NONE interchangeably.

WHEN ANY

An optional postscript that specifies that, when any previously specified WHEN condition in the SELECT is true, the statements that follow the WHEN ANY execute in addition to the statements that follow the first true condition.

ENDSEL|ENDSELECT

Reserved words that terminate the SELECT construct. If SELECT constructs are nested, the most recent unterminated SELECT construct is terminated by the first occurrence of ENDSEL or ENDSELECT. Each SELECT in a nested SELECT construct must have a corresponding ENDSEL or ENDSELECT.

Note: If the WHEN condition evaluates as unknown, the statements in the WHEN clause do not execute. The WHEN NONE clause executes if all WHEN conditions are unknown or false.

Examples

```
SELECT FIRST ACTION
         WHEN TOTAL_CHARGE > 250.00
             DO LARGE_PURCH
         WHEN CUSTOMER_CODE = 'P'
                 DO PREFERRD_CUSTMR
         WHEN TOTAL_CHARGE < 50.00
                 DO SMALL_PURCHASE
 WHEN OTHER
                 DO NO_DISCOUNT
 ENDSEL
```

The following procedure is equivalent to the format 1 example.

```
SELECT FIRST
  WHEN TRANS_CODE = 'A'
     DO ADD_RECORD_PROC
  WHEN TRANS_CODE = 'D'
     DO DEL_RECORD_PROC
  WHEN TRANS_CODE = 'P'
     DO PURCHASE_PROC
  WHEN TRANS_CODE = 'R'
     DO RECEIPT_PROC
  WHEN ANY
     DO LOG_TRANS
  WHEN OTHER
     DO INVALID_CODE
  ENDSEL
```

Format 3 of the SELECT statement selects one or more actions to take, based on all conditions found to be true. This executes each set of statements that follows each true condition.

```
SELECT EVERY [ACTION]
   WHEN  condition
      statements

   [WHEN  condition]
   [   statements  ] ...

     [     {NONE  } ]
     [WHEN {OTHER } ]
     [  statements  ]

   [WHEN ALL    ]
   [  statements]

   [WHEN ANY    ]
   [  statements]

 {ENDSEL    }
 {ENDSELECT}
```

[ACTION]

An optional reserved word that you can add for readability.

WHEN condition

Specifies a condition to test. For more information about valid conditions, see the "Procedure Definition Language Concepts and Language Elements" chapter. For every true condition, the statements that follow execute.

WHEN OTHER|NONE

An optional postscript that specifies that, when none of the conditions listed is true, the statements that follow the WHEN OTHER or WHEN NONE execute. The reserved words OTHER and NONE are interchangeable.

WHEN ALL

An optional postscript that specifies that, when all of the conditions are true, the statements that follow the WHEN ALL clause execute in addition to the statements following each true condition.

WHEN ANY

An optional postscript that specifies that, when any one condition is true, the statements that follow the WHEN ANY execute in addition to the statements that follow the true conditions.

ENDSEL|ENDSELECT

Reserved words that terminate the SELECT construct. If SELECT constructs are nested, the most recent unterminated SELECT construct is terminated by the first occurrence of ENDSEL or ENDSELECT. Each SELECT in a nested SELECT construct must have a corresponding ENDSEL or ENDSELECT.

If the WHEN condition evaluates as unknown, the statements in the WHEN clause do not execute. The WHEN NONE clause executes if all WHEN conditions are unknown or false.

Example

```
  SELECT EVERY ACTION
   WHEN NOT ITEM_NUMBER < 499
    DO ERROR_1
   WHEN NOT (DISCOUNT_CODE = 1 OR 3)
    DO ERROR_2 WHEN NOT ($NUMERIC(UNIT_PRICE)
    AND UNIT_PRICE > 0)
   DO ERROR_3
   WHEN ANY
    ADD 1 TO EDIT_FAILED
   WHEN NONE
ADD 1 TO EDIT_PASSED
   WHEN ALL
DO ALL_ERRORS
   ENDSEL
```

SET Statement

The SET statement transfers data from a source to a target. The SET statement does not modify the value of the source and the original data remains in the source after it is used to set the target.

There are six formats that you can use for the SET statement, depending on the type of source value used.

The SET Numeric Field statement has the following format:

```
                    {alphanumeric_expression}
SET numeric_field = {numeric_expression     }
                    {NULL                   }
```

Moves a value to an elementary numeric field according to the following rules:

- If both the source and target are numeric fields, the value is moved by alignment of an implicit decimal point, with truncation of low-order decimal digits (when necessary). If high order significant digits are lost, an overflow error condition is raised, and the error procedure, if coded, executes. See the error procedure section in this chapter.
- If the target is a date field, then the source cannot be an alphanumeric expression. However, you can set a date field using $INTERNAL-DATE with an alphanumeric argument.
- When a value from an alphanumeric source is moved to a non-date numeric target, the value of the source is first converted to numeric by applying the $NUMBER function (see the "Symbolic Debugger Commands" chapter). If the source contains non-numeric characters, an execution-time error occurs.
- If the source is the keyword NULL or evaluates to the null value, the target field must be nullable. It is set to the null value.

The SET Alpha Field statement has the following format:

```
                        {alphanumeric_expression }
                        {numeric_field           }
SET alphanumeric_field ={numeric_literal         }
                        {group                   }
                        {NULL                    }
```

Moves a value to an elementary alphanumeric field according to the following rules:

If both the source and target are alphanumeric fields, the value is moved as follows:

- If the source value is longer than the target, truncation occurs on the right.
- If the target is longer than the source value, the result is padded on the right with blanks.
- If the target is a variable length field, the length of the target becomes the length of the source, up to the maximum length defined for the variable length field.
- If the source is a variable length field, then its length is the length of the current value. This length is compared with the target as described above and is padded or truncated as required.
- If both source and target are the same length, an exact copy is made.

If the source is a non-alpha group, it is treated as an alphanumeric field whose length is the same as the size of the group. However, subordinate numeric fields are not converted. The hexadecimal values are simply moved.

You cannot specify restricted groups. For more information about non-alpha groups, see the "Procedure Definition Language Concepts and Language Elements" chapter.

PDF can convert low values moved to an alphanumeric panel field by using the $LOW function (or some other means) to special characters. See the *Creating Panel Definitions Guide* for more information. Fields filled with low values are considered empty. The next TRANSMIT fills this empty field with the output fill character when it is next displayed.

You cannot move a value directly from a date type source to an alphanumeric target. First, convert the date using an explicit $DATE or $STRING function.

When a value from a non-date numeric source is moved to an alphanumeric target, the data in the numeric source is first converted to alphanumeric by applying the $STRING function (see the "Symbolic Debugger Commands" chapter).

If you do not want to use the $STRING rules, you can specify a $EDIT function in a numeric expression as the source. Use the default PIC or specify one of the other edit patterns (see the "Symbolic Debugger Commands" chapter).

If the conversion results in a value that is longer than the target field after any leading blanks were removed, digits are truncated from the right to fit. For example, if NUM is a three-digit numeric field and STR is a two-character alphanumeric field, the expression results in STR having a value of '12' if NUM has a value of 123. If, however, NUM has a value of 12 or 1, STR has a value of '12' or '1', respectively.

```
SET STR = NUM
```

You cannot move an arithmetic expression or numeric function to an alphanumeric target without first moving it to a numeric target.

If the source is the keyword NULL or evaluates to the null value, the target field must be nullable. It is set to the null value.

The SET Group statement has the following format:

```
SET group1 = group2 {BY NAME    } [USING {$EDIT  } [RULES]
                    {BY POSITION} [      {$STRING}       ]
```

Moves data from one group to another. The following rules apply:

- Both the source and the target must be groups.
- Moving a value from each field in the source group to a field in the target group is subject to the rules for moving values described in Formats 1 and 2.

When values are moved from a non-alpha group to an alpha target, the default is to convert numeric data to alphanumeric by applying a $STRING function. You can specify to use a $EDIT function instead. See the options USING $EDIT RULES and USING $STRING RULES that follow.

BY NAME

Moves the value from each field in the source group to an identically named field in the target group if one exists. OCCURs in the respective groups (if any) must be compatible. Only values from fields that have the same names are subordinate to the respective groups and are elementary are moved. Redefinitions in the sending group are not eligible sources; however, redefinitions in the receiving group are eligible targets. A compile-time error message is issued if the number of occurrences does not match. A runtime error occurs if the number of occurs depending on or parameter field occurrences does not match.

For example, if the source group is:

Level	Field	Occur
1	A	
2	B	2
2	C	2

and the target group is:

Level	Field	Occur
1	X	
2	Y	2
3	B	
3	C	

then the statement,

```
SET X = A BY NAME
```

moves the values of the Bs and Cs in the source group to fields with corresponding names in the target group.

BY POSITION

Moves the value of the first elementary field in the source group item to the first elementary field in the target group item, regardless of its name; the second to the second, and so on. The group structures must be compatible as follows:

- Structures must have the same number of elementary fields.
- OCCURs values must be identical and at the same relative level.

You can test structures for compatibility by using the following steps:

1. Remove any OCCUR values from the high level group entries in both the source and target.
2. Remove any fields or groups with REDEFs in both the source and target group. If a group is removed, all its subordinate fields are also removed.
3. Remove each subordinate group entry except any that have OCCURs.
4. Renumber the levels for the fields that are left.

Note: No field values are moved unless the structures that remain after this test is applied are compatible.

USING $EDIT RULES

In a group move only, converts numeric fields to alphanumeric by applying a $EDIT function with an edit pattern of PIC 9(n), where n is the number of integer places in the field. The default is to use a $STRING conversion.

USING $STRING RULES(Default)

In a group move only, performs numeric conversion using the $STRING rules.

The statement SET Flag has the following format:

```
    SET flag = {TRUE  }
               {FALSE }
```

The item on the left side of the equal sign is a flag. The value on the right side of the equal sign must be the reserved word TRUE or FALSE.

The statement SET Condition has the following format:

```
SET condition_name = TRUE
```

Sets a condition name to the TRUE value.

The statement SET Function has the following format:

```
                      {numeric_expression   }
SET pseudo_function = {alphanumeric_literal}
                      {alphanumeric_field   }
```

You can only set the $RETURN-CODE function to a numeric_expression.

Examples

SET Numeric Field

```
SET CHARGE = QUANTITY_ORDERED * UNIT_PRICE
SET AMOUNT_RQRD = CURRENT_AMOUNT
SET PAY_AMT_NUMERIC = $NUMBER (PAY_AMT)
SET Z = ((Y + Z)*W)/(A - B)
SET L = A + ($REMAINDER(M,N) - $ROUND(J,1))/3.6
```

SET Alpha Field

```
SET ALPHA = $STRING (X,Y,Z)
```

SET Flag

```
SET EMP_NOT_VALID = FALSE
```

SET Condition

```
SET RED = TRUE
```

SET Function

```
SET $FIXED-MASK = '%'
```

Also see the MOVE statement examples in this chapter for examples that correspond to SET BY POSITION, SET BY NAME, and alpha group moves.

SET ATTRIBUTE/COLOR/XHIGHLIGHT Statement

The SET ATTRIBUTE/COLOR/XHIGHLIGHT statement resets certain terminal display characteristics for panel fields.

This statement has the following format:

```
SET [{ATTRIBUTE[S]}{attribute ...}]   [COLOR {color}]
    [{ATTR        }{'x ...'      }]   [      {'y'  }]

    [XHIGHLIGHT {xhighlight}]  [TEMP] ON fld_id [,fld_id]...
    [           {  'z'     }]
```

ATTRIBUTE

Specifies the general display characteristics of panel fields.

attribute

Specifies one or more of the following reserved words:

ALPHANUMERIC	CURSOR
ENSURE RECEIVED	HIGHLIGHT
INVISIBLE	LOWLIGHT
NUMERIC	PROTECTED
SKIP	UNPROTECTED

x

An alphanumeric literal in quotes that consists of the abbreviations of one or more of the above attributes. The abbreviation for each attribute is the first letter of the word or, for ENSURE RECEIVED, the first letter of the first word. For example H for HIGHLIGHT, L for LOWLIGHT, or E for ENSURE RECEIVED. Specify more than one attribute by concatenating the abbreviations as one quoted literal.

You can specify the attributes as follows:

- [UA]
- [UN] [H]
- [PA] [L]
- [PS] [I] [C] [E]

UA (A, U)

An unprotected field that can accept any characters. U, A, and UA are synonyms.

UN (N)

An unprotected field that can accept only numeric characters (0-9, decimal point, or comma). N and UN are synonyms.

PA (P)

A protected field that is not skipped. You cannot modify or delete a protected field.

PS (S, PN)

A protected field that the cursor skips over (cannot access). The cursor skips to *the* next unprotected field if the previous field is defined as PS or has an end-of-field mark. PN, PS, and S are synonyms.

- **H** A field displayed with high intensity characters.
- **L** A field that displays with *regular* (low) intensity.
- **I** An invisible field (input or *text*) where the characters do not display.
- **C** The field to contain the *cursor* when the panel displays.
- **E** A field that is treated as if it were entered on the current transaction. If a field has an attribute of E, it is assumed the user entered the value for that field, even if the value was entered on a previous transaction or was not entered at all (a default value).

See the *Creating Panel Definitions Guide* for more details on panel field attributes.

COLOR {color|Y}

Specifies the display color to use for the identified field.

color

One of the following reserved words:

BLUE	GREEN
NEUTRAL	PINK
RED	TURQUOISE
WHITE	YELLOW

Y

Represents an alphanumeric literal surrounded by quotes that consists of the initial character of one of the color specifications. That is, 'N' for NEUTRAL, 'B' for BLUE, and so on. You can only specify one color at a time.

XHIGHLIGHT

Specifies the type of extended highlighting used to display the identified field.

xhighlight

One of the following reserved words:

- BLINK
- NONE
- REVERSE
- UNDERSCORE

Note: If you specify UNDERSCORE, the field displays in reverse video.

Z

Represents an alphanumeric literal surrounded by quotes that consists of the initial character of one of the extended highlighting options. For example, 'N' for NONE, 'B' for BLINK, and so on.

TEMP

Limits the reset of the display characteristics to the next execution of a TRANSMIT statement. After the execution of the TRANSMIT statement, the value goes back to the setting defined for the field in the panel definition. The ENSURE RECEIVED attribute is implicitly temporary. All other characteristics must be explicitly specified as TEMP to limit the duration of the reset.

ON fld-id

Specifies the field or fields to receive the new characteristics, where fld-id is the identifier of the field to receive the characteristics. When C (cursor) is one of the attributes, you can specify only one fld-id.

When a SET ATTRibute statement executes in a program, storage is allocated containing ALL the attributes for the panel field. If the SET statement only specifies one attribute of a field, all the others assume the CA Ideal SET ATTRibute statement defaults, which are “UAL” and *not* the values of the original panel or any previous SET ATTRibute statement. If the SET ATTRibute statement has the TEMP option, then at the next TRANSMIT, the attribute storage is deleted and all attributes revert to the original panel field attribute values.

Ordinarily, a panel field is defined with a beginning field character and an end of field character, for example, a plus (+) and a semicolon (;). The end of field character is recognized as the beginning of a new field with an attribute of AUTOSKIP. If the layout of your panel definition includes the following characters, even though it looks like two fields with null characters between and after them, CA Ideal interprets it as four fields. The semicolon (end of field) says "start a new dummy field" with an attribute of AUTOSKIP, and when the user fills the first field, skip to the next field.

```
+                     ;  +      ;
```

To define a stopper field (a protected field that is not skipped), do not end the affected field with an end of field symbol. Code a one-byte field immediately following the affected field. Give this one-byte field an attribute of PAI.

The panel layout now looks like this:

```
+                    + ; +      ;
```

Overtyping the first field stops the cursor in the one-byte field and locks the terminal. To get to the next field, use the tab key.

If the ENSURED RECEIVED attribute is turned on in the panel field definition, you cannot turn it off with a SET ATTRIBUTE statement in the program.

You can specify attributes, color, and extended highlighting in any order, but you can specify each one only once.

Examples

The following two statements are equivalent and set FIELD_A to protected and skipped.

```
SET ATTRIBUTES PROTECTED SKIP ON FIELD_A
SET ATTRIBUTES 'PS' ON FIELD_A
```

The following statement sets field AMOUNT to highlighted, red, and underscored.

```
SET ATTRIBUTE HIGHLIGHT COLOR 'R' XHIGHLIGHT 'U' ON AMOUNT
```

The following statement sets fields MSG1 and MSG2 to skipped and highlighted, blue, and blinking.

```
SET ATTR 'SH' COLOR BLUE XHIGHLIGHT BLINK ON MSG1, MSG2
```

SUBTRACT Statement

The SUBTRACT statement decreases the value of numeric fields. You can use SUBTRACT as an alternative to the SET statement.

This statement has the following format:

```
         {numeric_field     }
SUBTRACT {numeric_literal   } FROM {numeric_field}
         {alphanumeric_field}      {date_field   }
         {date_field        }
```

For more information about *numeric_field*, *numeric_literal*, *alphanumeric_field*, and *date_field, s*ee the PDL Language Concepts topic in the "Procedure Definition Language Concepts and Language Elements" chapter.

During execution, both the source and the target fields must contain numeric values or a runtime error occurs.

SUBTRACT operands do not have to have the same decimal precision. When an expression with decimal places is subtracted from a field with an integer value, the subtraction is performed and an attempt is made to put the result into the receiving (FROM) field. If the value is too long, the decimal portion of the value is truncated. If the value that results from the truncation is still too long, a runtime error occurs.

You do not have to define the operands of a SUBTRACT statement with the same number of digits. However, an error occurs if the first operand is longer than the second operand or if the operation results in a value that is longer than the second operand.

Examples

```
SUBTRACT CHARGE FROM BALANCE
```

```
SUBTRACT 200 FROM NET_INCOME
```

TRANSMIT Statement

The TRANSMIT statement sends a previously defined screen and receives the data the user entered. When input is received from the user, the TRANSMIT statement automatically validates and edits all the data according to editing rules specified in the panel definition.

The *Creating Panel Definitions Guide* describes how to define and maintain screens. Several statements and built-in functions provide symbolic high-level panel processing. See the REFRESH and RESET statements and the $PANEL functions.

This statement has the following format:

```
TRANSMIT panel_name [REINPUT] [CLEAR]

     [ CURSOR AT {HOME            }]
     [           {field_identifier}]

     [ALARM]
```

panel_name

The one- to eight-character name of a panel.

REINPUT

Specifies that all fields on the panel for which $RECEIVED is true are still $RECEIVED true after the current TRANSMIT (when the panel is received again) regardless of whether you enter new values for those fields. REINPUT is used in situations where you are prompted repetitively until values for all required input fields are entered and validated. The entire panel is then processed in one action.

CLEAR

Resets all unprotected fields on the panel. All unprotected fields are sent with the output fill character and received back with the input fill character, unless they were modified. The statement TRANSMIT panel CLEAR is equivalent to the two statements:

- RESET panel
- TRANSMIT

CURSOR AT clause

Overrides the default position of the cursor when the panel is initially displayed.

Note: Positioning the cursor requires that the panel have at least one unprotected field.

HOME

Positions the cursor on the default field as defined. This option overrides a position set by a SET ATTRIBUTE statement.

- **field-identifier**

The identifier of the field where the cursor is initially displayed.

- **ALARM**

Activates the bell (if you have one) on your terminal when the panel is sent to the screen.

Data you enter is sent when you press the Enter key, when you press any PF key (except function keys assigned for HELP or CLARIFY), and on a scroll if specified. See the description of the parameter fill-in in the *Creating Panel Definitions Guide*.

Execution of subsequent TRANSMIT statements for the same panel sends the values of all fields the program or the user modified since the last TRANSMIT. For a fresh copy of the panel, use any of the following:

- REFRESH statement. All fields and their attributes are returned to their original values.
- RESET statement. All specified unprotected fields are reset to output fill characters.
- CLEAR option. All unprotected fields are reset to output fill characters.

You can initialize fields on a panel by moving values to those fields before the TRANSMIT statement is executed.

A TRANSMIT statement automatically causes a CHECKPOINT. See the *Creating Programs Guide* for details.

The BACKOUT statement can restore the state of the database to the most recent CHECKPOINT, BACKOUT, or TRANSMIT statement, or SQL COMMIT or ROLLBACK statement.

The combination of the following circumstances enters the error procedure or invokes the WHEN ERROR clause of the FOR construct if the following tasks are performed:

- A TRANSMIT statement executes in the scope of a FOR construct.
- A subsequent attempt is made to update the values of any fields for the same dataview record.
- Another task deleted a record or updated the values of any fields in that record between the TRANSMIT and subsequent updates.

The combination of the above results in a $ERROR-CLASS='DVW' that can be handled in either an error procedure or in the WHEN ERROR clause of the FOR construct.

For CA Datacom SQL ANSI Mode and DB2: If a TRANSMIT executes in the logical scope of a FOR construct for SQL access, you must define the corresponding object (table or view) with at least one unique index to update it. See the FOR EACH/FIRST statement (SQL Access) in this chapter.

Example

```
<<SEND_ADDPNL>> PROC
    TRANSMIT ADDPNLN
    statements :to process panel ADDPNL
    SET NEXT_PANEL = 'MAINPNL'
ENDPROC
```

Chapter 4: BuiltIn Functions

This chapter describes the CA Ideal functions.

$ABS Function

$ABS returns the absolute value of a numeric expression.

This function has the following format:

```
$ABS (numeric-expression)
```

The absolute value of the expression is returned. $ABS(X) is defined as follows:

```
If X<0, $ABS(X) is -X; else $ABS(X) is X.
```

Example

```
SET I = $ABS(J)     :If J is 6, I is 6.
                    :If J is -6, I is 6.
```

$ACCOUNTID Function

- You can set $ACCOUNT-ID to any four-character transaction identification or to the name of a field that contains a transaction ID.
- If you set $ACCOUNT-ID to an alphanumeric literal, the value must be surrounded by double or single quotes (' or ").

Important! You must first define any transaction ID set using the $ACCOUNT-ID in a SET statement in the CICS PCT. If the transaction identification does not exist, CA Ideal terminates and an error message from the TP monitor displays.

$ALPHABETIC Function

$ALPHABETIC evaluates to a value of True if a specified alphanumeric item is alphabetic; for example, it consists solely of the uppercase letters A through Z, lowercase letters a through z, and blanks. It evaluates to a value of False if the alphanumeric item is not alphabetic. If the value of the alphanumeric item is null, the function evaluates to a value of unknown.

This function has the following format:

`$ALPHABETIC(`*x*`)`

The identifier of an alphanumeric field or group.

Example

```
    IF $ALPHABETIC (ITEM-DESC) THEN
           DO PROC-ITEM-DESC
         ELSE
        DO ITEM-DESC-ERR
   ENDIF
```

If the identifier to test is nullable, test for null values (for example, IF identifier IS NULL ...) before testing with $ALPHABETIC. The function evaluates to Unknown if the value of the identifier is null.

A field of all spaces is considered alphabetic.

The site administrator can change valid alphabetic characters for a site in the PMS table PMSTBLS.

$APPL-ID Function

The $APPL-ID function returns the VTAM application identifier of the current region under CICS. The primary purpose is to identify the region for logging purposes.

The function returns an 8-byte character string, and is read-only.

This function has the following format:

`$APPL-ID`

If used in batch, it will return the value N/A.

$CALC Function

You can only use $CALC in the headings sections of report definitions. Its primary purpose is to combine report functions in report headings.

This function has the following format:

```
$CALC
```

You can calculate values in the control break and page footing specifications and the results included with the footing text. The $CALC function performs the calculations.

For example, you can use two columns in the CUSTOMER table, OPEN$ and YRTODATE, to compute the difference between the outstanding balance after processing the current orders and the previous outstanding balance. The calculation is coded for the control break footing on the Heading fill-in. The value is printed when a level-1 control break, based on state, occurs. The calculation is specified as:

```
$CALC ( $TOT(CUSTOMER.YRTODATE) - $TOT(CUSTOMER.OPEN$) )
```

When specifying $CALC, you can specify any valid arithmetic expression. The expression can include parenthesis, operators, report functions, field names, and numeric values. All fields and functions must be numeric. Field names must be unique names or labels defined in the primary detail group.

Valid operators include:

```
    +   -   *   /   $SQRT
```

Example

You can use $CALC to calculate the standard deviation. For example, to obtain the standard deviation for the value in OPEN$, a field containing the outstanding amount owed for each customer, a labeled detail expression is specified on the Detail fill-in as:

```
    OPEN2 = OPEN$ ** 2
```

The calculation in the footing on the Heading fill-in is specified as:

```
    $CALC ($SQRT($OCC(OPEN$) * $TOT(OPEN2) - $TOT(OPEN$) * TOT(OPEN$))
       / $OCC(OPEN$))
```

$CHARTOHEX Function

$CHAR-TO-HEX is an alphanumeric function that displays non-printable fields. It returns the hexadecimal value of the specified identifier or literal. If you are using the function to set or move values, the length of the receiving field should be at least twice the size of the sending field to avoid truncation.

This function has the following format:

```
              {alpha-expression}
              {num-field       }
$CHAR-TO-HEX( {num-literal     } )
              {flag            }
```

alpha-expression

Defines an alphanumeric expression.

num-field

Specifies the numeric field.

num-literal

Specifies the numeric literal.

flag

Specifies a flag.

For more information about alphanumeric expressions, numeric fields, numeric literals, and flags, see the PDL Language Concepts topic in the "Procedure Definition Language Concepts and Language Elements" chapter.

Note: Hexadecimal representation of fields on the PC can differ from hexadecimal representation on the mainframe. For this reason, results from logical tests based on hexadecimal representation can differ between the PC and the mainframe.

Example

Assume that B = 'ABCD' sets A to 'C1C2C3C4'. If B was defined in working data as a five-byte field, the value of A would be 'C1C2C3C440'.

```
SET A = $CHAR-TO-HEX(B)
```

$COUNT Function

$COUNT returns the current number of iterations of the referenced FOR FIRST, FOR EACH, or LOOP construct.

This function has the following format:

```
$COUNT(label)
```

label The 1- to 15-character label of a FOR EACH or LOOP construct.

- Before any execution of the referenced construct, 0 is returned. After the end of execution of such a construct, the final total number of iterations is returned.
- PROCESS NEXT increments $COUNT and iterating the loop.
- QUIT does not increment $COUNT.
- A loop that terminates due to a VARYING clause increments $COUNT before testing whether the THRU limit was exceeded.

Example

```
    <<DEPT>>  FOR EACH WORKER
WHERE EMP-DEPT = 'J'
       : number and list the
       : names of all workers
       : in department 'J'
MOVE $COUNT(DEPT) TO WCOUNT
LIST WCOUNT, EMP-NAME
ENDFOR
```

$CURRENTTRANID Function

This function returns the four-character transaction ID currently in effect.

This function has the following format:

```
    $CURRENT-TRAN-ID
```

In *CICS,* the $CURRENT-TRAN-ID function returns the current transaction ID.

$CURSOR Function

This function evaluates to a Boolean value of True or False, depending on whether the cursor is in the designated field or row in a panel. A value of False is returned if the panel was not transmitted.

This function has the following format:

```
         {pnl-grp(pnl-row) }
$CURSOR ({field-identifier })
```

pnl-grp

Specifies the identifier of a repeating group field on a panel. You must define this field for the panel. It can be nullable.

pnl-row

Specifies the name of a field or a literal that indexes the repetitions of the repeating group. You cannot specify a nullable field as pnl-row. The value of $CURSOR is True if the cursor is in any field on the specified row.

field-identifier

Specifies the identifier of an elementary field tested. You must define this field for the panel.

$DATE Function

$DATE returns an alphanumeric value of either the specified date or the current date in the specified format. See the SET COMMAND SESSION OPTIONS and SET COMMAND DATEFOR commands in the *Command Reference Guide*. The defaults are those defined at runtime.

This function has the following format:

```
$DATE [(['date-pattern'][,DATE=input-date])]
```

$DATE alone returns the current date in the default date format, which you can set in session or site options.

'date-pattern'

Displays the sequence of characters (maximum 30), in quotes, that represents the format in which components of the date (day, month, and year) are returned. The default pattern is the format displayed in the current site option fill-in. The notations that you can specify for each date component are shown in the following table.

Component Notation	**Meaning**	**Assuming January 10, 1993**
YEAR	Year in full	1993
YY	Year without century	93
Y	Year without decade	3
MONTH	Month spelled out (uppercase)	JANUARY
LCMONTH	Month spelled out (initial letter uppercase)	January
MON	Month abbreviation (uppercase)	JAN
LCMON	Month abbreviation (initial letter uppercase)	Jan
MM	Month number, with leading zero if necessary	01
M	Month number with no leading zero	1
DD	Day with leading zero if necessary	10
D	Day with no leading zero	10
DDD	Julian day, numeric day of the year (1-366)	010
WEEKDAY	Day spelled out (uppercase)	SUNDAY
LCWEEKDAY	Day spelled out (initial letter uppercase)	Sunday
DAY	Day abbreviation	SUN
LCDAY	Day abbreviation (initial letter uppercase)	Sun
ISOWEEK	International Standards Organization date format	1993-01

Any characters except uppercase alphabetics in the date pattern remain unchanged.

The site administrator for each site defines the actual text indicated by the keywords MONTH, LCMONTH, MON, LCMON, WEEKDAY, LCWEEKDAY, DAY, and LCDAY in the PMS table PMSTBL.

input-date

Displays the date used as input to the function. The input-date can specify a literal, the name of a field containing the date, or the current date (that is, the date at runtime). If no DATE= clause is specified, the default input-date is the current date. The input-date must be between 2000 B.C. and 9999 A.D.

You can enclose the entire input-date in parentheses. The input-date is specified as follows:

```
{$TODAY                                                   }
{date-field                                               }
{'literal'                                                }
{alpha-date [,TEMPLATE='alpha-input-pattern'] [,BASE=yyyy]}
{                                                         }
{num-date [,TEMPLATE='num-input-pattern'] [,BASE=yyyy]    }
{MONTH=month, DAY=day, YEAR=year [,BASE=yyyy]             }
```

$TODAY

Specifies a numeric function that returns the CA Ideal internal integer date for the current date (that is, the date at runtime). Each time $TODAY is encountered, it calls the operating system.

date-field

Specifies the name of a date field defined in working data or parameter data.

'literal'

Defines a six-character alphanumeric literal in the form 'yymmdd' (or in the format specified in a TEMPLATE clause described in the TEMPLATE=topic). Trailing blanks are ignored.

The numbers represented can be from 000101 to 991231. In this range, *yy* can be from 00 to 99, *mm* can be from 00 to 12, and *dd* can be from 01 to 31. The maximum value for *dd* depends on the value of *mm* and *yy*. The day specified must exist.

alpha-date

Specifies an alphanumeric field containing a value in the default format yymmdd, or in the format specified by the accompanying TEMPLATE clause. Trailing blanks are ignored.

TEMPLATE='alpha-input-pattern'

Defines the pattern of the date in the accompanying field or literal. The input pattern is built in much the same way as the output date pattern. Up to 30 characters represent components and notation of the date being read by the function. However, several keywords available in the output date-pattern are not available here. The notations for specifying the format of each input date component are shown in the following table.

Component Notation	Meaning	Assuming January 10, 1993
YEAR	Year in full	1993
YY	Year without century	93
MONTH	Month spelled out (uppercase)	JANUARY
LCMONTH	Month spelled out (initial letter uppercase)	January
MON	Month abbreviation (uppercase)	JAN
LCMON	Month abbreviation (initial letter uppercase)	Jan
MM	Month number, with leading zero if necessary	01
M	Month number with no leading zero	1
DD	Day with leading zero if necessary	10
D	Day with no leading zero	10
DDD	Julian day, numeric day of the year (1-366)	010
*	Mask character, meaning that any character except a numeric digit may appear in this position	
blank	Zero or more blanks may be entered in this position	

M and D must be followed by an asterisk (*) or blank or used at the end of the pattern. The site administrator for each site defines the actual text indicated by the keywords MONTH, LCMONTH, MON, and LCMON in the PMS table PMSTBLS. For more information, see the *Working in the Environment Guide*.

num-date

A numeric field containing a numeric date. This value is converted to alphanumeric format and interpreted using the default format yymmdd or the format specified by the accompanying TEMPLATE clause.

TEMPLATE='num-input-pattern'

Defines the pattern of the date in the accompanying numeric field. Up to 30 characters represent the formats of the components of the date being read by the function. The input pattern is built in much the same way as the date pattern, but only the keywords in the following table are available.

Component Notation	Meaning	Assuming January 10, 1993
YEAR	Year in full	1993
YY	Year without century	93
MM	Month number, with leading zero if necessary	01
DD	Day with leading zero if necessary	10
D	Day with no leading zero	10
DDD	Julian day, numeric day of the year (1-366)	010

BASE=yyyy

Specifies a four-digit number as the base year from which the century of the input year is determined (the default is 1900). The input date must include the year in the format *yy* (for example Mar 3, 84). If you include this clause, the following algorithm defines the year's century:

```
If yy >= the last two digits of the base year
   use the first two digits of the base year as the century.
Else
```

- use the first two digits of the base year plus one as the century.
- **Input-date Clause:** DATE='010286',TEMPLATE='MMDDYY',BASE=1950 ...

  Date Specified: January 2, 1986
- **Input-date Clause:** DATE='010201',TEMPLATE='MMDDYY',BASE=1950 ...

  Date Specified: January 2, 2001

- **Input-date Clause:** DATE='010250',TEMPLATE='MMDDYY',BASE=1950 ...
- **Date Specified:** January 2, 1950

  MONTH=month

  DAY=day

  YEAR=year-Specifies an input date when the month, day, and year are separate numeric literals or numeric fields. Each operand can be a number or field. For example:

  ```
  MONTH=12,DAY=7,YEAR=year-field
  ```

  If you omit one or more, the input date is interpreted as explained in the first note below. If the year is a two-digit value, you can specify a BASE= year for the century. If the year is a four-digit value, the BASE= year is ignored. A three-digit day (DDD) is not valid.

If the century is required for the output format or for $INTERNAL-DATE format, the input date must be supplied either by a 4 digit YEAR or 2 digit YY with a BASE year.

If you do not specify the month, day, or year of the input date, the component is interpreted as follows:

```
month=1 day=1 year=current
```

Only the parts of the six-character input date that are actually required by the 'date-pattern' are edited for validity. The remaining characters must be represented, but their values are not tested. For example, the following date functions include an input date with a month of 23:

```
$DATE('YY',DATE='852307') is valid.
$DATE('MM',DATE='852307') is in error.
```

Examples

The following example converts a Gregorian International date to a Julian date. W_JUL_DAT and W_GREG_DAT are both type X fields.

```
SET W_JUL_DAT =$DATE('YYDDD',DATE=W_GREG_DAT, TEM ='YYMMDD')
```

Assume that it is January 10, 1993.

$DATE	**Example**	**Specifics**
$DATE	('MONTH DD, YEAR')	JANUARY 10, 1993
$DATE	('M/DD/YY')	1/10/93
$DATE	('DD/MM/YY')	10/01/93
$DATE	('MON. DD, YEAR')	JAN. 10, 1993
$DATE	('DD MON YEAR')	10 JAN 1993

$DATE	Example	Specifics
$DATE	('MON. 12, 1988')	JAN. 12, 1988
$DATE	('MON. DD, YY',DATE='880326')	MAR. 26, 88
$DATE	('YEAR',DATE='85XXXX')	1985
$DATE	('DD', DATE='XXXX31')	31
$DATE	('MONTH',DATE='XX05XX')	MAY

If you want a period after an abbreviation, you must include it as part of the literal. You must also enter spacing between components.

$DAY Function

$DAY returns the numeric value for the day of the month (1-31), either for the current date or for the date in the specified date field.

This function has the following format:

```
     [($TODAY)    ]
$DAY [(date-field)]
```

$DAY alone returns the day of the month for the current day ($TODAY).

For a complete explanation of date-fields, see PDL Language Concepts topic in the "Procedure Definition Language Concepts and Language Elements" chapter.

Example

Assume W_DATE is a type D field. The following gives the value of the last day of the previous calendar month with respect to W_DATE.

```
W_DATE - $DAY(W_DATE)
```

$EDIT Function

$EDIT returns an alphanumeric value by editing the given field or literal according to the specified pattern.

This function has the following format:

```
       {num-literal}
$EDIT ({num-field  }  [,PIC='edit-pattern'])
       {alpha-field}
```

num-literal

Specifies the numeric literal to edit.

num-field

Specifies the numeric field to edit.

alpha-field

Specifies the alphanumeric field to edit.

PIC='edit-pattern'

Specifies the format of the field or literal. The format can be any valid pattern and must be enclosed by quotes. In addition, an L in the first character of the edit-pattern left-justifies the result.

The maximum length of an edit pattern is 30 characters. Spaces found anywhere in the edit pattern are not suppressed.

The PIC= clause is required for an alpha-field. If you do not specify a PIC= clause for a numeric field, the default edit pattern is PIC 9(*n*), where *n* is the number of integer places. For example, for a numeric field NUM_FLD with four integer and two decimal places, the function,

```
$EDIT(NUM_FLD)
```

is equivalent to,

```
$EDIT(NUM_FLD,PIC='9999')
```

If you assume that I is an unsigned numeric with three digits and two decimals, and J is a signed numeric field with six digits, for

```
SET I = 34.5
$EDIT (I)
```

the result is '034', and for

```
SET J = -123
$EDIT (J)
```

the result is '123'.

Edit Pattern Rules

The following table shows the edit pattern rules. In this table,

- *Source* means the data as it is contained in the dataview field, panel field, working data field, and so on, and whose name is specified as the field to edit.
- Lowercase v represents the position of an assumed decimal point.
- Edit patterns can be condensed by using multipliers. For example, you can specify the expanded pattern,

  ```
  ZZZ,ZZZ.99
  ```

  in the edit pattern as:

  ```
  Z(3),Z(3).9(2)
  ```

Category and Meaning		Data in Source	Edit Pattern	Resulting Display
Pattern characters for alphanumeric data				
X	Alphanumeric character	STATE	X(5)	STATE
	Any other character represents itself	AB1234	XXX-XXX	AB1-234
Pattern characters for numeric data				
L	Left-justify before output	123	LZ (3),ZZ9	123
9	Unsuppressed numeric digit	123	999	123
Z	Zero suppression	v12	ZZZ.99	.12
*	Asterisk replacement	1v23	***9.99	***1.23
,	Comma	002234v56	ZZZ,ZZZ.99	2,234.56
/	Slash	123083	99/99/99	12/30/83
B	Blank space	123083	99B99B99	12 30 83
0	Zero	123	99900	12300
.	Decimal point	030v99	ZZZ.99	30.99
-	Minus sign, fixed right	-23v45 23v45	9 (2).99- 9 (2).99-	23.45- 23.45
-	Minus sign, fixed left	-23v45	-9 (2).99	-23.45
-	Minus sign, floating	-v23	- - .99	-.23
+	Plus sign, fixed right	67v89	9 (2).99+	67.89+
+	Plus sign, fixed left	67v89	+9 (2).99	+67.89
+	Plus sign, floating	00v67	++.99	+.67

Category and Meaning		Data in Source	Edit Pattern	Resulting Display
CR	Credit symbol, right	-25v00 25v00	9 (2).99CR 9 (2).99CR	25.00CR 25.00
DB	Debit symbol, right	-13v00 13v00	9 (2).99DB 9 (2).99DB	13.00DB 13.00
$	Dollar sign, fixed	004v00	$z9.99	$ 4.00
$	Dollar sign, floating	001v23	$$$.99	$1.23
<>	Enclosed negative numbers in parentheses, fixed	-23v45	<9,999.99>	(0,023.45)
<>	Enclosed negative numbers in parentheses, floating	-23v45	$<<<<.99>	$ (23.45)

$EMPTY Function

This function evaluates to a Boolean value of True or False, depending on whether a panel field contains a value.

This function has the following format:

```
        {pnl-grp(pnl-row)      }
$EMPTY ({panel field-identifier })
```

pnl-grp

The identifier of a repeating group field on a panel. You must define this field for the panel. It can be nullable.

pnl-row

The name of a field or a literal that indexes the repetitions of the repeating group. You cannot specify a nullable field as pnl-row. If all fields in the row identified by this value are empty, $EMPTY returns a value of True.

panel field-identifier

The identifier of an elementary field tested. You must define this field for the panel. The field can be nullable.

$EMPTY tests True if the field or row had no initial value assigned at layout and any of the following occurred:

- The program or user did not modify the field or row.
- The user erased the field or row by pressing the EOF key.
- The field or row was cleared with a TRANSMIT panel-name CLEAR statement.
- A RESET statement cleared the field or row or the entire panel.
- The panel was cleared with a REFRESH panel-name statement.
- The field or row contains null values. (The function is equivalent to an IS NULL conditional expression.)

$EMPTY tests False if any of the following occurred:

- The field or row was laid out with an initial value and not modified.
- The program or the user entered a value into the field or row.
- The field or row was assigned an ensure received (E) attribute and then transmitted.

$ENTERKEY Function

This function evaluates to a Boolean value of True or False, depending on whether you pressed the Enter key.

This function has the following format:

```
$ENTER-KEY [(panel-name)]
```

panel-name

When there is more than one currently active panel, you can specify a panel name. The default panel is the latest panel received.

$ENVIRONMENT Function

This function returns a string value containing the type of environment the program is running in (batch or online).

This function has the following format:

```
$ENVIRONMENT
```

$ERRORCLASS Function

This function returns a three-letter code that identifies the general category of the most recent error.

Note: $ERROR functions only return meaningful data in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

This function has the following format:

```
$ERROR-CLASS
```

Example

```
    <<ERROR>> PROC
    SELECT $ERROR-CLASS
    WHEN 'DVW'
       DO DVW-ERROR
    WHEN  'PGM'
       DO PGM-ERROR
    WHEN OTHER
       DO OTHER-ERROR
    ENDSELECT
             PROCESS NEXT DVW
ENDPROC
```

$ERRORCLASS and $ERRORTYPE Codes

CLASS	TYPE	DESCRIPTION
ARI		**ARITHMETIC ERRORS**
	DVZ	An operation resulted in an attempt to divide by zero.
	EXP	An exponent violates the rules for exponents, for example, is not a numeric integer value or exceeds 999.
	OFL	An overflow condition occurred.
	SQR	An attempt was made to find the square root of a negative value.
	UNS	An attempt was made to assign a negative value to an unsigned numeric field.
DVW		**DATAVIEW ERRORS**
	ARN	There is an error in the assignment of a report or dataview (VSE only).
	DB2	There is an error in a DB2 database access. (See $ERROR-DB2-PLAN and $ERROR-DVW-STATUS.)

CLASS	TYPE	DESCRIPTION
	DVW	There is a CA Datacom/DB error in a Native Access, SQL access, or sequential file access dataview, or there is an error in VSAM access. (See also $ERROR-DVW-STATUS and $ERROR-DVW-INTERNAL-STATUS.)
	D50	There is an error in the date conversion for CA Datacom/DB native or SQL access.
	D71	The record is not found in the UPDATE clause of a WHERE condition.
	D72	The record is not found in the DELETE clause of a WHERE condition
	D73	The required row ID is missing
	D74	There is an invalid ODO (occurs depending on) item in the VSAM file.
	D75	All 16 cursors are open.
	D76	The DB2 column is not in date format
	D77	The transaction ID is not in the RCT.
	D78	There is a date and time mismatch in SQL.
	D80	The PLAN module is not found, DYNAMIC SQL=NO.
	D81	The program is not in the PLAN, DYNAMIC SQL=NO.
	D82	There is a compiled plan date error, DYNAMIC SQL=NO.
	D83	The PLAN module is not found, RUN SQL is static.
	D84	The program is not in the PLAN, RUN SQL is static.
	D85	There is a compiled plan date error, RUN SQL is static.
	D86	The RCT exit is not PLNPGME=@IADRCTX.
	Q17	There is a CA Datacom/DB database error with an SQL code of -117.
	Q18	There is a CA Datacom/DB dictionary error with an SQL code of -118.
	SQL	There is an error in a CA Datacom SQL database access. See also $ERROR-DVW-STATUS.
	VBO	The VSAM base cluster or one of the alternate paths could not be opened successfully during batch execution. See also $ERROR-DVW-DBID.
	VBP	An IBM VSAM error occurred. See also $ERROR-DVW-STATUS and $ERROR-DVW-INTERNAL-STATUS.
	VCO	There is no FCT entry open for this VSAM ddname.
	VCP	An IBM VSAM error occurred while running in CICS. See also $ERROR-DVW-STATUS and $ERROR-DVW-INTERNAL-STATUS.
	VCV	A VSAM dataview with variable-length records requires a variable-length FCT entry in CICS.

CLASS	TYPE	DESCRIPTION
	VRB	CA Ideal cannot resume a browse when the access key is not unique
	VRL	The length of an individual VSAM record is not the expected length.
	VRN	An invalid number was used in a WHERE clause for a Relative Record data set (RRDS).
	VVL	The dataview definition compiled with the program does not agree with the actual VSAM file opened by the program.
FTL		**FATAL ERRORS**
	D70	The index used as a row ID no longer exists or is no longer unique.
	F60	An attempt was made to access a report facsimile that is not defined in the program resources.
	F61	An attempt was made to run a program that was not compiled since the last edit.
	F62	An attempt was made to access a facsimile that is not available.
MIS		**MISCELLANEOUS ERRORS**
	A34	An invalid value was assigned to a report date.
	A35	A value assigned to a report page number exceeds the maximum number allowed.
	A36	A value assigned to a report page size exceeds the maximum size allowed.
	A37	SET $PLAN was issued in a logical unit of work.
	ARD	An invalid disposition was specified for a report.
	ATM	An attempt to access a panel that the user is not authorized to access.
	ATP	An attempt to access a program that the user is not authorized to access.
	BPA	An attempt to access a panel in a batch run is invalid.
	DBC	A DBCS detach error occurred.
	D40	An invalid value was assigned as a HEX-TO-CHARACTER field.
	D41	The PAD length specified exceeds the internal limits.
	D42	The input date value is not a positive integer.
	D43	The date format contains invalid characters.
	D44	The date format exceeds the length limit.
	D45	The week value of the date is invalid.
	D46	The month value of the date is invalid.
	D47	The day value of the date is invalid.

CLASS	TYPE	DESCRIPTION	
	D48	The year value of the date is invalid.	
	D49	The DATE value exceeds the maximum length.	
	D51	The PAD length is less than the original length.	
	DTE	An invalid value was specified for a $DATE or a $TIME function.	
	SDV	An attempt was made to read a sequential file dataview online.	
NUM		**NUMERIC ERRORS**	
	NUM	A numeric field contains an invalid numeric value.	
PGM		**PROGRAM ERRORS**	
	IQP	A QUIT or PROCESS NEXT was used incorrectly.	
	NID	An attempt was made to reference a non-ideal program that does not exist in the load library.	
	PGM	An attempt was made to recursively enter an active program.	
	PRO	An attempt was made to recursively enter an active procedure.	
REF		**REFERENCE ERRORS**	
	NUL	An attempt was made to reference a null field where null values are not allowed.	
	PAT	An attribute of a parameter does not match the corresponding attribute of the field where it is passed.	
		Subtypes:	
		01	Run parameter longer than specified
		02	Copied panel (should never occur)
		03	Occurs not matched
		04	Type not matched
		05	Subprogram parameter redefined
		06	Parameter structures unequal
		07	Displacement not identical
		08	Numeric class, and so on, not identical
		09	Numeric precision not identical
		10	Nullable not matched
		11	Variable not matched
		12	Alpha not matched

CLASS	TYPE	DESCRIPTION	
		13	Alpha exceeds the declared length
		14	Occurs exceeds max
		15	Occurs not identical
		16	ODO displacement not equal
		17	ODO not matched
		18	Panel field not identical
	PIU	An attempt was made to pass an INPUT parameter to a field designed to receive an UPDATE parameter.	
	PKY	A parameter keyword does not match another parameter keyword.	
	REF	An attempt was made to reference a field that was never accessed or is no longer available.	
	UPD	An attempt has been made to update a field that is not updateable.	
	V07	One of the VSAM dataview functions ($RRN, $RBA, or $REC-LENGTH) was used when its value is not defined.	
SEQ		**SEQUENCE ERRORS**	
	ADB	An attempt was made to ASSIGN a dataview that is active.	
	ARS	An attempt was made to ASSIGN a report that is active.	
	DEL	A DELETE was specified that is invalid, for example, was not issued from a FOR FIRST or FOR EACH construct or was applied to a dataview that is not updateable.	
	FOR	An attempt was made to nest a FOR on the same dataview.	
SUB		**SUBSCRIPT ERRORS**	
	GRP	The number of OCCURs does not match between a sending and receiving group.	
	ODO	An ODO (occurs depending on) value exceeds the maximum allowed.	
	SST	The start parameter of a substring is less than 1 or the length parameter is less than 0.	
	SUB	The form of a subscript is invalid; that is, it is less than 1 or greater than the number of occurrences.	
SYS		**SYSTEM ERRORS**	
	CVR	A system error occurred.	
	SYS	A serious system error occurred.	
	USR	A user error specified in the DESCRIPTION field of a message.	

CLASS	TYPE	DESCRIPTION
	INT	An internal error specified in the DESCRIPTION field of a message.
USR		**USER REQUESTED ERRORS**
	USR	The application invoked the DO ERROR statement.

$ERRORCONSTRAINTNAME Function

This function provides the constraint name when the error is a referential integrity violation.

Note: $ERROR functions only return meaningful data in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

For CA Datacom/DB native access, the $ERROR-CONSTRAINT-NAME function returns the constraint name for a database return code of 94, with a subcode of 31, 34, or 35. For all other errors, this function returns a value of N/A.

For SQL access, the $ERROR-CONSTRAINT-NAME function returns the constraint name from the error message text for SQL codes -175, -176, and -260.

This function has the following format:

```
$ERROR-CONSTRAINT-NAME
```

$ERRORDB2PLAN (DB2 Only)

This alphanumeric function returns the seven-character DB2 application plan in effect for the dataview in error. If the dataview is not in error or if the dataview was a CA Datacom/DB or sequential dataview, the function returns N/A.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORDESCRIPTION Function

This function returns an alphanumeric message of up to 80 characters that describes the most recent error.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORDVWDBID Function

This function returns a database ID that identifies the most recent error.

For CA Datacom/DB Native Access

This alphanumeric function returns the three-character database ID, with leading zeros, of the dataview in error. If the dataview is not in error (or if it was a sequential or SQL dataview), the function returns N/A.

For VSAM Dataviews

This function returns the eight-character ddname of the path in error.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORDVWINTERNALSTATUS Function

This function returns a code that identifies the most recent error.

For CA Datacom/DB Native Access

This function returns the CA Datacom/DB internal return code. This code is returned only in the presence of a dataview error and only for CA Datacom/DB Native Access dataviews. See the *CA Datacom/DB* Messages *and Codes Guide* for an explanation of the conditions that return this code.

For VSAM Dataviews

When $ERROR-CLASS returns a code of DVW and $ERROR-TYPE returns a code of VCP, the value of $ERROR-DVW-INTERNAL-STATUS can be interpreted as follows:

This function has the following format:

```
ffff-xxxxxx
```

ffff

Specifies the four-character hexadecimal EIBFN function code.

xxxxxx

Specifies the six-character hexadecimal CICS EIBRCODE.

When $ERROR-CLASS returns a code of DVW and $ERROR-TYPE returns a code of VBP, the value of $ERROR-DVW-INTERNAL-STATUS can be interpreted as follows:

This function has the following format:

iiff-xxccrr

ii

The CA Ideal internal function code.

ff

The two-character VSAM function code that CA Ideal assigns.

xx

The two-character hexadecimal VSAM return code.

cc

The two-character hexadecimal VSAM component code.

rr

The two-character hexadecimal VSAM reason code.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORDVWSTATUS Function

This function returns a code that identifies the status of a dataview request (only when $ERROR-CLASS returns a code of DVW).

$ERRORTYPE DVW

CA Datacom/DB Native Access Dataview or Sequential File Dataview:

Type	Description
I1	End of volume reached (SEQ).
I2	Errors found in the data of this record (DB).
I3	Record integrity problem occurred because a user modified a record that another user was processing (DB, VSAM).
I4	More than 16 SEQUENTIAL files used (SEQ).
I5	z/OS: Missing DD statement for the SEQUENTIAL file (SEQ, VSAM). VSE: Invalid ASSGN card (SEQ, VSAM).
I6	Actual record length in a sequential file greater than the record length in the dataview (SEQ).
I7	BLOCKSIZE of the file defined in the VPE file table too small (VSE SEQ).
I8	No entry defined for the file in the VPE file table (VSE SEQ).
I9	The length of the right-hand operand was greater than that of the left-hand operand when a CONTAINS was applied (DB).

If a code does not start with an I, see the return codes in the *CA Datacom/DB Messages and Codes Guide* for an explanation of the condition that issued the code. All codes are issued as a result of errors, but can instead be issued for exceptional conditions such as a record interlock.

$ERRORTYPE SQL or DB2

For SQL Dataview:

The SQLCODE from the SQLCA is returned as a character string. The value can be between -999 and +999. Negative numbers represent errors.

$ERRORTYPE Vxx

For VSAM Dataviews:

- When $ERROR-TYPE returns a code of VVL, $ERROR-DVW-STATUS returns the 11-character validation error name (DATASETTYPE, RECLENGTH, KEYLENGTH, or KEYOFFSET).
- When $ERROR-TYPE returns a code of VCP or VCV, $ERROR-DVW-STATUS returns the 11-character VSAM CICS condition name.
- When $ERROR-TYPE returns a code of VBP or VBO, $ERROR-DVW-STATUS returns a code that can be interpreted as follows:

This function has the following format:

```
mm-nnn
```

mm

Specifies the two-character VSAM return code in decimal digits.

nnn

Specifies the three-character VSAM reason code in decimal digits.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORNAME Function

This function returns the name of error field, subscript, dataview, procedure, or program that depends on $ERROR-TYPE. The $ERROR-NAME value can be up to 65 characters in length.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORPGM Function

This function returns the name of the program that contains the most recent error. In a complex application, an error in a subprogram can cause errors in calling programs. All errors are logged in the RUNLIST if LIST ERROR executes with each Error procedure for the application's subprograms.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORPROC Function

This function returns the name of the procedure that contains the most recent error. Use this function with $ERROR-PGM and $ERROR-STMT to identify the line number, procedure, and statement number where the error occurred.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORSTMT Function

This function returns the sequence number of the statement that contains the most recent error.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORSUBSCRIPT Function

This function returns the position of the subscript (1, 2, or 3) on a subscript error. Otherwise, N/A is returned.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORTYPE Function

This function returns a three-letter code that identifies the specific type of error. See $ERROR-CLASS for the type codes associated with each class.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$ERRORVALUE Function

This function returns the value of a numeric field when non-numeric characters moved into the field result in a runtime error. The $ERROR-VALUE value can be up to 30 characters in length.

When the ERROR PROCEDURE is invoked because a numeric field contains non-numeric data, you can use $ERROR-VALUE to see which positions contain the non-numeric data. The resulting string of $ERROR-VALUE shows a ? where non-numeric data was found. For example, if the string 12A45B was moved into a numeric field, the error procedure is invoked and $ERROR-VALUE contains 12?45?. The question marks indicate the positions where non-numeric data was found.

Note: $ERROR functions return meaningful data only in the error procedure or in a procedure or subprogram invoked by the error procedure. In the case of a database error, $ERROR functions can also return meaningful data in the scope of a WHEN ERROR clause. You do not have to pass the $ERROR functions as parameters.

$FINALID Function

$FINAL-ID is used in a SET statement to dynamically assign a transaction identification to schedule when the CA Ideal session is terminated explicitly with an OFF command or implicitly when SET RUN QUITIDEAL is set to Y. You can only use this function as the target of a SET or MOVE statement. You cannot interrogate it.

To transfer to a blank CICS screen at termination, set $FINAL-ID to NONE.

This function has the following format:

```
$FINAL-ID
```

Do not start transaction IDs $FINAL-ID uses with the CICS reserved character C.

You can set $FINAL-ID to a four-character transaction ID, to the name of a field that contains a transaction ID, or to the reserved transaction identification NONE. Do not use transaction IDs used with $FINAL-ID with $ACCOUNT-ID. It should be associated with the program SC00INIT in the PCT.

If $FINAL-ID is set to an alphanumeric literal or to NONE, the value must be surrounded by double or single quotes (' or ").

Example

```
SET $FINAL-ID='PAYR'
```

$FIXEDMASK Function (CA Datacom/DB Native Access)

$FIXED-MASK (or $FIX-MASK) in a SET statement to assign a character to use as a mask character when a where-condition in a FOR construct for a CA Datacom/DB native access dataview uses a CONTAINS or NOT CONTAINS operator.

A mask character blanks out one character in an expression. You can use the mask character any number of times in the expression so that a search can be made for other characters in the expression, regardless of context.

This function has the following format:

```
$FIXED-MASK
```

or

```
$FIX-MASK
```

- You can set $FIXED-MASK to any single non-blank character. The default mask character is the asterisk (*).
- The total of the number of mask characters and search characters must not exceed the total field length. If the total of the number of mask characters and search characters is less than the total field length, then the search string can be found anywhere in the expression.
- The default mask character is in effect until it is reset with $FIXED-MASK. Once the mask character is set (either explicitly or by default), that character remains in effect for the entire run or until it is explicitly reset. Therefore, if a calling program contains a where condition that uses a mask character and a subprogram resets the mask character and then returns to the calling program, the mask character remains reset and applies to any CONTAINS clauses in the calling program. This holds even if a different mask character was used in the calling program.

Examples

Assume a six-character alphanumeric field called FIELDA and a mask character that was allowed to default to * (asterisk). A conditional expression that contains the following clause searches for values of field FIELDA that have an A in position 2 and a BC in positions 4 and 5 respectively, regardless of what characters are present in positions 1, 3, and 6:

```
WHERE FIELDA CONTAINS '*A*BC*'
```

Assume a six-character alphanumeric field called FIELDB and a mask character that was set to _ (underscore). A conditional expression that contains the following clause searches for values of FIELDB that have AB in positions 2 and 3, 3 and 4, or 4 and 5:

```
WHERE FIELDB CONTAINS '_AB_'
```

Assume a five-character field called FIELDC with a value of ABCDE and a mask character set to #. For the following test, the results are as shown in the following table:

```
WHERE FIELDC CONTAINS compare-string
```

Compare-string	Result
'A#C#E'	True
'#ABC#'	False
'##BC#'	False
'CDE'	True
'B#C'	False
'#CD'	True
'#AB'	False
'DE#'	False

$HEXTOCHAR Function

$HEX-TO-CHAR is an alphanumeric function that returns the display representation of the specified hexadecimal expression.

This function has the following format:

```
$HEX-TO-CHAR(alpha-expression)
```

The alphanumeric input must have an even number of valid hexadecimal characters.

This function is the counterpart to $CHAR-TO-HEX.

Example

Assume that a one-byte hexadecimal field is coded in the database for each sales region. The following example sets a one-character alphanumeric working data field SALES.REGION to the non-printing hexadecimal code.

```
  <SET-PROC>
SELECT
   WHEN PNL.REGION EQ 'EAST'
      SET SALES.REGION = $HEX-TO-CHAR('09')
   WHEN PNL.REGION EQ 'WEST'
      SET SALES.REGION = $HEX-TO-CHAR('0A')
ENDSEL
```

$HIGH Function

The $HIGH function returns the highest value in the alphanumeric collating sequence.

This function has the following format:

```
$HIGH
```

- You can use $HIGH as a source field in a MOVE or SET statement. It assumes the same length as its associated target.
- You can use $HIGH in a conditional expression as the object of a comparison. It assumes the same length as the value to which it is compared.
- $HIGH has a length of 1 when used as the argument of $STRING or in a LIST statement.
- You can use $HIGH as an alphanumeric expression in the WHEN clause of a SELECT construct. $HIGH assumes the length of the SELECT subject.

Example

Assume that A, B, and C are 16-character alphanumeric fields.

```
SET A = $HIGH       : returns 16 X'FF
    SET B = $LOW        : returns 16 X'00'
    SET C = $SPACES     : returns 16 X'40'
```

$HOST-ID Function

The $HOST-ID function returns the value of the user's identity as provided to the host environment (CICS or batch). It may have a different value from that of $USER-ID or $USER-NAME according to the way sign-on was performed.

This function has the following format

```
$HOST-ID
```

The function returns an 8-byte character string, and is read-only.

$INDEX Function

$INDEX locates the left-most position in an alphanumeric expression where a search string can be found. $INDEX returns 0 if the search string is not found in the expression. It returns a value of NULL if the expression or the search string evaluates to NULL.

This function has the following format:

```
$INDEX (alphanumeric-expression,SEARCH=substring)
```

alphanumeric-expression

Defines the string to scan for the first occurrence of the substring.

Substring

Specifies the alphanumeric identifier, literal, or alpha-group that is the substring to find. This expression must be surrounded by delimiters if it is an alphanumeric literal. It can be a nullable field.

This function is especially useful for extracting a substring with the $SUBSTR function, as shown in the following example.

Example

Assume that A is a 17-character alphanumeric expression, B is a five-character alphanumeric expression, and I is a numeric field.

```
SET A = 'THESE THREE WORDS'
    SET I = $INDEX (A,SEARCH='THREE')    :result is 7
    SET B = $SUBSTR (A,START=I,LENGTH=5) :result is 'THREE'
    SET B = $INDEX (A, SEARCH= 'TTT')    : result is 0
```

$INITTRANID Function

This function returns the four-character transaction ID of the transaction that accesses CA Ideal.

This function has the following format:

```
$INIT-TRAN-ID
```

$INTERNALDATE Function

This is a numeric function that returns the CA Ideal internal integer date for the specified input date. The internal date represents the number of days difference between the input date and December 31, 1900.

This function has the following format:

```
$INTERNAL-DATE(input-date)
```

input-date

Specifies the date used as input to the function. The input-date can specify a literal or the name of an alphanumeric or numeric field containing the date. Specify the input-date as follows:

```
{date-field                                          }
{'literal'                                           }
{alpha-date [,TEMPLATE='alpha-input-pattern'][,BASE=yyyy]}
{                                                    }
{num-date [,TEMPLATE='num-input-pattern'] [,BASE=yyyy]   }
{MONTH=month, DAY=day, YEAR=year [,BASE=yyyy]            }
```

The input date is interpreted using the default format *yymmdd* and the BASE value or the format specified in the TEMPLATE clause when it includes a 4 digit year. For explanations of the input date options, see the section on the $DATE function in this chapter

Example

The following returns the internal integer format of the date stored in PAY_DATE in the format *yymmd.* If the year is less than 50, then the century is 20. If the year is greater than 50, then the century is 19.

```
$INTERNAL-DATE(PAY_DATE,TEMPLATE='YYMMDD',BASE=1950)
```

The following function returns the internal integer format of the date literal 19940415 (April 15,1994), namely, 31151.

```
$INTERNAL-DATE('19940415',TEMPLATE='YEARMMDD')
```

$KEY Function

An alphanumeric function that returns an indicator of the last key pressed for the last panel transmitted or the specified panel.

This function has the following format:

```
$KEY [(panel-name)]
```

$KEY

Returns the name of the last key pressed for the specified panel. If no panel-name is specified, $KEY returns the name of the last key pressed for the last panel transmitted.

panel-name

The name of the panel tested.

This function returns a variable-length indicator. The values returned are:

ENTER

N/A (panel not transmitted)

PF01

PF02

.

.

.

PF*nn*

Example

The NOTIFY statement displays a message on the panel message line. The following statement checks that the last key pressed was PF1-9 (for example, was not Enter, and that N/A was not returned). If not, it displays a message specifying the invalid key typed by the user:

```
IF $KEY(PANEL-A) GE 'PF01' AND $KEY(PANEL-A) LE 'PF09'
  DO PROCESS-PANEL
ELSE
  NOTIFY $KEY ' IS NOT VALID FOR THIS TRANSACTION.
'ENDIF
```

$LENGTH Function

$LENGTH is a numeric function that returns the length of the specified alphanumeric expression.

This function has the following format:

```
$LENGTH(alpha-expression)
```

alpha-expression

An identifier comprising alphanumeric fields, groups, and functions. If you specify a variable length alpha field, the current length (as opposed to the maximum defined length) is returned.

Example

Assume that PARM_A is a variable length alphanumeric field defined in a calling program with current length 15 and a parameter in a subprogram defined with length 20. The following function in the called subprogram returns a length of 15.

```
$LENGTH(PARM_A)
```

Assume VAR is a variable length field defined with a maximum length of 11, and FIX is a fixed length field, also of 11.

```
SET VAR = 'VAR'
SET N = $LENGTH(VAR)
LIST N
3 will be returned.

SET FIX = VAR
SET N = $LENGTH(FIX)
LIST N

11 will be returned.
```

$LOW Function

This function returns the lowest value in the alphanumeric collating sequence.

This function has the following format:

```
$LOW
```

You can use $LOW as a source field in a MOVE or SET statement. It assumes the same length as its associated target.

You can use $LOW in a conditional expression as the object of a comparison. It assumes the same length as its associated comparand.

You can use $LOW as an implied comparand in a SELECT construct. It assumes the length of its associated source comparand.

$LOW has a length of 1 when used as the argument of $STRING or in a LIST statement.

Example

Assume that A, B, and C are 16-character alphanumeric fields.

```
SET A = $HIGH       : returns 16 X'FF'
SET B = $LOW        : returns 16 X'00'
SET C = $SPACES     : returns 16 X'40'
```

$MONTH Function

$MONTH is a numeric function that returns the month number (1-12) either for the current date or for the date in the specified date field.

This function has the following format:

```
           [($TODAY)    ]
$MONTH [(date-field)]
```

$MONTH alone returns the month for the current date ($TODAY).

$NETWORKID Function

The $NETWORK-ID function on the mainframe returns the eight-character network node name.

This function has the following format:

```
$NETWORK-ID
```

In CICS**,** the value this function returns depends on whether the terminal is in VTAM.

- If the terminal is VTAM, the function returns the VTAM LU name.
- If the terminal is not VTAM but is MRO, the function returns the system ID and terminal ID of the Terminal Owning Region (TOR).
- In all other circumstances, the function returns low values.

$NUMBER Function

$NUMBER returns a numeric value by converting a given alphanumeric value.

This function has the following format:

$NUMBER(*name*)

name

An identifier of an alphanumeric field or group whose value must consist only of an optional sign (+ or -) followed by numerals with one optional decimal point. Leading or trailing blanks are acceptable but are ignored in the input value.

This function is not required since conversion to a numeric format is performed automatically when necessary. However, for program clarity, use this function explicitly.

When numeric conversion is performed automatically, the compiler issues a message that an alphanumeric item is used in a numeric context and that a number-compatible form is assumed.

Examples

```
SET X = '-234.56'
    SET J = $NUMBER (X)  :result is numeric form of -234.56

    SET Y = '    -123.86
    SET K = $NUMBER (Y)  :result is numeric form of -123.86
```

$NUMERIC Function

$NUMERIC evaluates to a value of True if the value of the specified identifier can be used in CA Ideal arithmetic. It evaluates to a value of False if the use of the identifier causes an arithmetic or conversion error. It evaluates to a value of Unknown if the value of the identifier is null.

This function has the following format:

`$NUMERIC(`*name*`)`

name

Specifies an identifier of a numeric or alphanumeric field or of an alpha group.

For $NUMERIC to return a value of True for an alphanumeric item, the item must consist only of an optional sign (+ or -) followed by numerals with one optional decimal point. Leading or trailing blanks are permitted, but are ignored in the alphanumeric input value.

For $NUMERIC to return a value of True for a numeric item, the item must have a valid numeric internal representation. (An example of a numeric field that does not return True is a numeric packed or binary field that redefines an alphanumeric field.)

You can only test 31 significant digits, exclusive of leading zeros.

The $NUMERIC function tests whether a value can be converted to a numeric data type. It does not test whether the value is actually numeric. Therefore, when testing a panel field for a numeric entry, use the function $PANEL-FIELD-ERROR.

If the identifier to test is nullable, test for null values (for example, IF identifier IS NULL ...) before testing with $NUMERIC. The function evaluates to Unknown if the value of the identifier is null.

Example

```
IF $NUMERIC (ORDER_FORM.QUANTITY) THEN
    SET EXTENSION = $NUMBER(ORDER_FORM.QUANTITY) * PRICE
    ELSE
    DO NON_NUM_QTY
    ENDIF
```

$OPSYSTEM Function

This function returns a string value containing the operating system the program is running under (MVS/XA or DOS).

This function has the following format:

```
$OPSYSTEM
```

Note: The compatibility with existing applications requires that these values do not change, so z/OS systems will return a value of *MVS/XA* and *VSE* systems will return DOS.

$PACKAGESET Function

$PACKAGESET returns the value of the DB2 special register CURRENT PACKAGESET. The value of $PACKAGESET is stored with $PLAN. You can use the SET $PACKAGESET statement to specify a logical identifier for the next package to use. Unlike $PLAN, this is the actual package name.

This function has the following format:

```
$PACKAGESET
```

As the source of a SET statement or in a condition, $PACKAGESET returns the package name most recently set in a CA Ideal application.

As the target of a SET statement, you can use $PACKAGESET to select a package name that can instruct DB2 to change to a new package. You can set $PACKAGESET to any one- to eight-character package identification or to the name of a field that contains a package identification. If $PACKAGESET is set to an alphanumeric literal, you must surround the value with double or single quotes (" or '). For example:

```
SET $PACKAGESET = 'PAYPKG'
```

- $PACKAGESET can be interrogated or set when a program executes in dynamic mode, but the function only has an affect on the application when it runs in static mode.
- You can set a session package name using a SET RUN PACKAGESET command. The SET $PACKAGESET statement overrides this setting for the current application.
- If a package name was not set in the application, the function returns the name set for the session by SET RUN PACKAGESET.
- If a package name was not set in the application and a name was not set for the session, the function returns ID????DV. For example, the following tests true if the name PAYPKG was selected in the most recent SET $PACKAGESET statement or if a SET $PACKAGESET statement was not executed in a SET RUN PACKAGESET command.

  ```
  IF $PACKAGESET = 'PAYPKG' ...
  ```

You can execute the SET $PACKAGESET statement only before the first SQL statement in a logical unit of work. That is, executing SET $PACKAGESET at any point except before the first SQL statement at the beginning of a CICS transaction or before the first SQL statement following a database Commit causes a runtime error.

The first SQL statement can be embedded SQL, SQL generated by a FOR construct for a DB2 dataview, or SQL in a non-ideal subprogram. A Commit can be a PDL TRANSMIT, CHECKPOINT or BACKOUT statement, or an SQL COMMIT or ROLLBACK statement.

If an application calls a non-ideal subprogram that executes SQL statements and if that SQL cam be the first in a logical unit of work, then you must specify Y (yes) for the Access DB2 field on the non-ideal program IDE panel.

Changes made to the package name by a package name exit do not affect the value of $PACKAGESET. $PACKAGESET reflects the value set by statements in the current application.

Examples

This code selects PAYPKG as the CA Ideal application's package name for the embedded SQL that follows it.

```
CHECKPOINT
SET $PACKAGESET = 'PAYPKG'
EXEC SQL ...END-EXEC
...
```

The following procedure saves the package name that was selected in a previous procedure in a Working Data field SAV-PKG. It sets GETPKG as the package name to use with the FOR construct that follows it, commits its database modifications, and resets the package name before returning.

```
<<GET>> PROCEDURE
SET SAV-PKG = $PACKAGESET
SET $PACKAGESET = GETPKG
FOR EACH DB2-DVW
   ...
ENDFOR
CHECKPOINT
SET $PACKAGESET = SAV-PKG
ENDPROC
```

$PAD Function

$PAD is an alphanumeric function that returns the string that results from filling an alphanumeric expression with the specified character on the left, right, or both ends, to the length specified.

This function has the following format:

```
                                     [RIGHT= {'a'     }]
$PAD(alpha-expression,LENGTH=num-exp,[CENTER={a-field }])
                                     [LEFT=            ]
```

alpha-expression

Any alphanumeric expression.

num-exp

A numeric expression indicating the length of the string produced by adding fill characters to the expression. It must be defined with integer digits only. If any other kind of numeric expression is specified or if the value is not a whole number, the integer portion is used.

RIGHT|CENTER|LEFT

The RIGHT= clause adds the specified character to the right of the expression. The CENTER= clause adds the specified character equally to the right and left. The LEFT= clause adds the specified character to the left of the expression. You cannot specify more than one clause. If you do not specify a clause, the default is RIGHT=' ' (pad with trailing blanks). If you specify a clause, its keyword can be abbreviated to the first three characters (RIG=, LEF=, CEN=).

'a'

A single-character, alphanumeric literal used as the fill character.

a-field

A one-byte alphanumeric field containing the fill character. (This cannot be a variable length field.) You cannot use $HIGH, $LOW, and $SPACE.

- The function does not change the alphanumeric expression. It uses it to build the expression that it returns. Thus, a return value longer than the original expression is possible.
- If the length specified is less than or equal to the length of the alphanumeric expression, the expression is returned unchanged.
- If you specify CENTER= and the specified length minus the length of the alphanumeric expression is an odd number, then the extra byte is placed on the right.

Examples

Assume that IDENT is a four-byte alphanumeric field containing 'ABCD'. The following function results in a nine-byte result containing 'ABCDbbbbb', where b is a blank.

```
$PAD(IDENT,LENGTH=9)
```

Using the same input field, IDENT,

```
$PAD(IDENT,LENGTH=4)
```

returns 'ABCD'.

The following example converts five-digit zip codes in DVW.ZIP.CODE to nine-digit zip codes by padding to the right with zeros. ZIP_NINE is defined in working data as a nine-byte alphanumeric field.

```
SET ZIP_NINE = $PAD(DVW.ZIP.CODE,LENGTH=9,RIGHT='0')
```

The string returned by $PAD is an intermediate result. When the intermediate result is moved to a field, the final result depends on the field type and length. For example, given the following fields:

- FIX_FLD-Type X, length 10, initial value 'ABC'
- VAR_FLD-Type V, length 10, initial value 'ABC'.

The following function returns the string 'ABC????bbb', where 'b' is a blank.

```
SET FIX_FLD = $PAD(VAR_FLD, LENGTH =7, RIGHT='?')
```

The following statement sets FIX_FLD to 'ABCbbbbbbb'. Because the length specified (7) is less than the length of the alphanumeric expression (FIX_FLD is 10 characters long), the expression is returned unchanged.

```
SET FIX_FLD = $PAD(FIX_FLD,LENGTH=7, RIGHT='?')
```

On the other hand, the following statement sets VAR_FLD to 'ABCbbbbbbb'.

```
SET VAR_FLD = $PAD(FIX_FLD, LENGTH=7, RIGHT='?')
```

The statement following sets VAR_FLD to 'ABC????'.

```
SET VAR_FLD = $PAD(VAR_FLD, LENGTH=7, RIGHT='?')
```

A nested example, such as that shown, sets FIX_FLD to 'ABC????bbb'.

```
SET FIX_FLD = $PAD($TRIM(FIX_FLD),LENGTH=7, RIGHT='?')
```

Intermediate results in a nested example are treated the same as variable length moves.

$PANELERROR Function

$PANEL-ERROR is a Boolean function that evaluates to True when an input error is detected in an input field of a panel. As long as one field contains erroneous data, $PANEL-ERROR is True. If more than one field is detected with erroneous data, $PANEL-ERROR remains true until the user corrects all the fields in the panel or until the application modifies all of the erroneous fields. You can modify fields with the following statements: MOVE, SET, RESET, REFRESH, or RELEASE.

$PANEL-ERROR is only meaningful when the panel parameter Edit-rule error procedure is set to A (application). If this parameter is set to C (Clarify), the CLARIFY function prevents detectable errors before returning control to the application. See the *Creating Panel Definitions Guide* for more information.

This function has the following format:

```
$PANEL-ERROR[(panel-name)]
```

panel-name

Specifies the panel containing the fields to test for erroneous data. When you omit a panel name, the function applies to the last panel transmitted.

Example

In this example, the panel is retransmitted until all the errors detected in the input panel fields are corrected.

```
    LOOP
TRANSMIT PANEL-A REINPUT
    WHILE $PANEL-ERROR
ENDLOOP
```

$PANELFIELDERROR Function

This function returns a number indicating the error status of the specified panel field.

This function assists in error processing but you can only use it if the edit-rule error procedure option of the panel parameter fill-in was specified as A for application. For more information, see the *Creating Panel Definitions Guide*.

This function has the following format:

```
$PANEL-FIELD-ERROR(field-name)
```

field-name

The field on the panel tested.

The following chart lists the values that can be returned and explains what they mean.

Number	Description
0	No error.
1	A required field is missing.
2	Non-numeric data is detected in a numeric field.
3	Field content is outside the range specified as the minimum and maximum values allowed for the field.
4	An invalid check digit was specified.
5	The field entry does not have the required number of decimal places.

Number	Description
6	A field specified as must-fill was not filled.
99	Reserved for future use. If this code is returned, call CA Ideal Technical Support.

Examples

```
 LOOP
SET USER-ERROR = FALSE    :  Defined in working data
TRANSMIT PANEL-A REINPUT
SELECT   WHEN $PANEL-FIELD-ERROR(FIELD-1) = 3
   SET MESSAGE-FIELD = “FIELD-1 IS OUTSIDE OF RANGE”
WHEN $PANEL-FIELD-ERROR(FIELD-99) = 1
   SET MESSAGE-FIELD = “REQUIRED FIELD IS MISSING”
WHEN $PANEL-FIELD-ERROR(FIELD-2) = 1   :  Not received
   SET FIELD-2 = 'default-value'
WHEN PNL-FDL1 NOT = PNL-FLD2 * 2
   SET MESSAGE-FIELD = “UNSUPPORTED FUNCTION”
   SET USER-ERROR = TRUE
ENDSELECT
WHILE $PANEL-ERROR(PANEL-A) OR USER-ERROR
  ENDLOOP
```

In this example, a panel is analyzed for field entries that violate the panel definition's validation rules or the application's rules. The panel is retransmitted until all input errors are corrected. The $PANEL-ERROR function controls the exit from the loop for panel definition rules, while USER-ERROR, defined in working data, controls the exit for application rules. USER-ERROR is set to false before each TRANSMIT because the user can change field contents. It is re-evaluated whenever the panel is processed.

If a value is not entered in Field-2, the application supplies a default value. If this was the last panel definition violation, $PANEL-ERROR becomes false. The application handles a test that is not supported (PNL-FLD2 * 2) and, if it fails, sets USER-ERROR to true, which requires the user to correct the entry.

Example of Help Processing:

```
<<A-PNL>>
  LOOP                     : User help and error analysis
     SET USER-ERROR = FALSE: Defined in working data
     TRANSMIT PANEL-A REINPUT
     IF $PF1
       TRANSMIT A-HELP
       PROCESS NEXT A-PNL
     ENDIF
     SELECT
       . . .  (User error analysis)
     ENDSELECT
  WHILE $PANEL-ERROR(PANEL-A) OR USER-ERROR
ENDLOOP
```

In this example, the end user has the option of invoking a help facility (PF1). Again, $PANEL-ERROR and USER-ERROR control the exit from the TRANSMIT loop. The REINPUT option is used with the TRANSMIT statement so that you do not need to reenter required fields each time through the loop.

Example of User Escape:

```
<<MENU-SEL>> PROCEDURE
      LOOP
      TRANSMIT MENU
      SELECT MENU-OPTIONS
      WHEN “1”
         DO PROCESS-ORDER
      WHEN “2”
       . . .
    ENDLOOP

    <<PROCESS-ORDER>> PROCEDURE
      LOOP
      SET USER-ERROR = FALSE     : Defined in working data
      TRANSMIT ORDER-SC
      . . .  (User error analysis)
      IF $PF24
           RELEASE PANEL ORDER-SC
         QUIT PROCESS-ORDER
      ENDIF
       WHILE $PANEL-ERROR OR USER-ERROR
   ENDLOOP
   . . .  (Process order)
ENDPROC
```

In this example, the PROCESS-ORDER procedure provides PF24 as a way for the user to escape from the current panel and go back to the MENU-SEL procedure. This is helpful in preventing deadlock if incorrect data was entered and the user does not know the correct data.

$PANELGROUPOCCURS Function

This function returns the number of occurrences of a repeating group in a panel. You can define panels with a variable number of occurrences of a group of fields, the number of which is determined by the number of lines that remain in the region of the screen during application execution. However, there can only be one repeating group per panel.

This function assists in error processing but can only be used if the edit-rule error procedure option of the panel parameter fill-in was specified as A for application. For more information, see the *Creating Panel Definitions Guide*.

This function has the following format:

```
$PANEL-GROUP-OCCURS (panel-name)
```

panel-name

The name of the panel for which the number of occurrences is requested.

$PF Function

This function evaluates to a Boolean value of True or False, depending on whether a program function key was pressed.

This function has the following format:

```
 {$PF1 }
 {$PF2 }
 { .   } [(panel-name)]
 { .   }
 {$PFn }
```

panel-name

When there is more than one currently active panel, you can specify a panel name. The default panel is the latest panel transmitted.

Example

```
    TRANSMIT PANELX
SELECT
WHEN $PF1
   DO PROC-PF1
WHEN $PF2
   DO PROC-PF2
WHEN $ENTER-KEY
   DO PROC-ENTER
WHEN OTHER
   IF $RECEIVED(ORDER-NO)
      DO PROC-NORMAL-ORDER
   ELSE
      MOVE REPROMPT-MSG TO ORDER-FORM.MSG
      SET ATTRIBUTES HIGHLIGHT
         ON ORDER-FORM.MSG, ORDER-NO
   ENDIFENDSEL
```

$PLAN Function

$PLAN returns the application plan name most recently set in an application. You can use the SET $PLAN statement to specify a logical identifier for the next plan to use. This is a logical identifier, not necessarily the actual plan name. This level of indirection lets you avoid hard coding actual plan names in your programs (which requires compilation for any change to plan names).

This function has the following format:

```
$PLAN
```

As the source of a SET statement or in a condition, $PLAN returns the plan name most recently set in a CA Ideal application.

As the target of a SET statement, you can use $PLAN to select a plan name that can instruct DB2 to change to a new plan. You can set $PLAN to any one- to eight-character plan identification or to the name of a field that contains a plan identification.

If $PLAN is set to an alphanumeric literal, you must surround the value with double or single quotes (“ or '). For example:

```
SET $PLAN = 'PAYPLAN'
```

Online, CA Ideal applications can switch plan names only if DB2 Version 2 is installed and the RCT entry for the current transaction ID specifies PLNEXIT=YES and PLNPGME=@IADRCTX, the RCT exit CA Ideal supplies.

In batch, CA Ideal applications can switch plans under either release of DB2.

For more information about selecting plans in CA Ideal and setting up the RCT, see the *Administration Guide*.

- You can set a session plan name using a SET RUN PLAN command. The SET $PLAN statement overrides this setting for the current application. If a plan name was not set in the application, the function returns the name set for the session by SET RUN PLAN.
- If a plan name was not set in the application and a name was not set for the session, the function returns IDPLANDV. For example, the following tests true if the name PAYPLAN was selected in the most recent SET $PLAN statement or if a SET $PLAN statement was not executed in a SET RUN PLAN command.

  ```
  IF $PLAN = 'PAYPLAN' …
  ```
- You can execute the SET $PLAN statement only before the first SQL statement in a logical unit of work. That is, executing SET $PLAN at any point except before the first SQL statement at the beginning of a CICS transaction or before the first SQL statement following a database commit causes a runtime error.
- The first SQL statement can be embedded SQL, SQL generated by a FOR construct for a DB2 dataview, or SQL in a non-ideal subprogram. A commit can be a PDL TRANSMIT, CHECKPOINT or BACKOUT statement, or an SQL COMMIT or ROLLBACK statement.
- If an application calls a non-ideal subprogram that executes SQL statements and if that SQL can be the first in a logical unit of work, then you must specify Y (yes) for the Access DB2 field on the non-ideal program IDE panel.
- You can use the $ERROR-DB2-PLAN function in an error procedure to return the actual plan name currently in effect.
- Changes made to the plan name by a plan name exit do not affect the value of $PLAN. $PLAN reflects the value set by statements in the current application.

Examples

This code selects PAYPLAN as the CA Ideal application's plan name for the embedded SQL that follows it.

```
CHECKPOINT
SET $PLAN = 'PAYPLAN'
EXEC SQL …END-EXEC
…
```

The following procedure saves the plan name that was selected in a previous procedure in a working data field SAV-PLAN. It sets GETPLAN as the plan name used with the FOR construct that follows it, commits its database modifications, and resets the plan name before returning.

```
<<GET>> PROCEDURE
SET SAV-PLAN = $PLAN
SET $PLAN = GETPLAN
FOR EACH DB2-DVW
…ENDFOR
CHECKPOINT
SET $PLAN = SAV-PLAN
ENDPROC
```

$PROGRAM Function

The $PROGRAM function returns the eight-character name of the current program. See also $SYSTEM and $VERSION.

This function has the following format:

```
$PROGRAM
```

- You can use this function for security testing or accounting purposes.

$RBA Function

The $RBA function accesses ESDS VSAM files by the relative byte address. The function returns the current relative byte address in numeric form. You can use the $RBA function in the WHERE clause of a FOR statement to select records, as the source operand in SET statements, and in expressions.

This function has the following format:

```
$RBA[(dataview-name)]
```

dataview-name

The name of a VSAM dataview that defines an ESDS file. The dataview name is required for this function except when it is used as the left-hand operand in a WHERE clause. You cannot specify the dataview name for this function in the left-hand operand of the WHERE clause.

- You cannot use the $RBA function as the target of a SET or MOVE statement.
- The $RBA function is accessible only when the dataview fields are accessible. This means that you cannot use the function before the execution of a FOR statement. If you try to reference this function before the FOR statement is executed, a run-time error occurs.
- The starting relative byte address must be exactly equal to the $RBA of a valid record.
- You cannot use this function in a FOR NEW construct since a record does not have a relative byte address until it is written.
- The first record in an ESDS file has a relative byte address of 0.

Examples

In the following example, the $RBA function is accessed after the FOR NEW processing is completed to determine the relative byte address of the last record added.

```
    FOR NEW PSS-MASTER
    statements
ENDFOR
SET NEW-ADDRESS = $RBA(PSS-MASTER)
```

In the next example, a table of relative byte addresses determines the starting address.

```
    FOR FIRST PSS-MASTER
    WHERE $RBA = RBA-TABLE(REC-NO)
       statements
ENDFOR
```

$RECEIVED Function

This function evaluates to a Boolean value of True or False, depending on whether a value was received for the specified field. You can issue the $RECEIVED function for panel fields or rows or for parameters passed from a RUN command or a CALL statement.

This function has the following format:

```
           {pnl-grp(pnl-row) }
$RECEIVED ({field-identifier })
           {pnl-grp(pnl-row) }
```

pnl-grp

The identifier of a repeating group field defined for a panel.

pnl-row

The identifier of a field or a literal used to index the repeating group. If a value was received for any field on the specified row, the $RECEIVED function returns a value of True.

field-identifier

The identifier of an elementary panel field or level-1 parameter field being tested. This field must be defined for the panel or in the parameters.

For parameters, you can only test level-1 fields or groups. It is not possible to omit parameters at other levels.

$RECEIVED for fields passed from a RUN command or a CALL statement is set to True when the level-1 parameter was provided. If a level-1 parameter value was not received, all level-1 parameters defined on subsequent lines of the parameter section are also missing. See the Creating Programs Guide.

$RECEIVED for panel fields evaluates to the value True in three cases:

- The user actually modified the value of the specified field on the most recent TRANSMIT of the panel (including retyping existing characters or pressing ERASE-EOF).
- The user modified the value of a specified field on a previous TRANSMIT and REINPUT was specified for each subsequent TRANSMIT of the panel.
- The specified field was defined with an attribute of E (ensure received) in one of two ways:
  - With a SET ATTR 'E' statement
  - Defined as ensure received when the panel was specified

The value of $RECEIVED is not reset as long as panel errors remain.

$RECLENGTH Function

This function is used with VSAM records to determine the length of the current record. You can use the $REC-LENGTH function as the source of data in any statement, provided that a record was read before the statement is encountered.

This function has the following format:

```
$REC-LENGTH(dataview-name)
```

dataview-name

The name of a VSAM dataview. The dataview name is required for this function.

In the scope of a FOR EACH or FOR FIRST statement, the value of the $REC-LENGTH function is the length of the current record. If $REC-SEGMENT (see the $REC-SEGMENT function that follows) is reset in the FOR statement, the value of $REC-LENGTH changes to the length of the record with the new variable segment. When used with an OCCURS DEPENDING ON clause, the value of $REC-LENGTH actually changes after the ENDFOR when the record was written.

In the scope of a FOR NEW statement, the value of the $REC-LENGTH function is the maximum length of the record until a new length is set by changing the value of the $REC-SEGMENT function or when the ENDFOR is reached.

You can only use the $REC-LENGTH function as the source of data in a CA Ideal statement. You cannot use it as the left operand in a WHERE clause of a FOR statement.

You can use the $REC-LENGTH function whenever the fields in a dataview can be used. The function has no value before a record is read, after a record is deleted, and if a WHEN DUPLICATE or WHEN NONE condition is encountered. An attempt to access the function in these circumstances causes a run-time error.

When updating an ESDS data set, you cannot actually change the length of an existing record; however, the $REC-LENGTH function returns the value that is correct for the $REC-SEGMENT used to write the record. When the new length is shorter than the original record length, the unused part of the record is filled with binary zeroes. If the new length is longer than the original length, a run-time error occurs.

Example

The following example shows how you can use $REC-LENGTH to determine the record type of a variable-segment ESDS record that does not include a fixed-length segment. A SELECT construct tests the length of each input record and executes the appropriate procedure for that record type.

```
    FOR EACH PSS-MASTER
    SELECT FIRST ACTION
      WHEN $REC-LENGTH(PSS-MASTER) = 110
        DO PROCESS-UPGRADE
      WHEN $REC-LENGTH(PSS-MASTER) = 72
        DO PROCESS-LOAN
      WHEN $REC-LENGTH(PSS-MASTER) = 95
        DO PROCESS-CHECK
      WHEN ANY
        DO PROCESS-SUMMARY
      WHEN NONE
        DO PROCESS-ERRORS
    ENDSELECT
ENDFOR
```

$RECSEGMENT Function

This function is used with variable-segment VSAM records to specify which record type (which variable segment) to write. Although the function is primarily intended for use in the FOR NEW construct, you can also use it in a FOR EACH or FOR FIRST construct to change a record to a new record type.

This function has the following format:

```
$REC-SEGMENT(dataview-name)
```

dataview-name

The name of a KSDS or ESDS VSAM dataview. The dataview name is required for this function.

You can only use the $REC-SEGMENT function as the target of a SET statement, with the source (the value on the right of the equal sign) being the name of a level-2 field that is one of the variable segments defined in the dataview. Since the field name, not the field content, is the value set, the field name must be enclosed in single or double quotes (' or “).

- The $REC-SEGMENT function actually controls the record length, not the layout. You are responsible for using the correct field names. When moving data to variable-segment records, specify the level-2 group name. Do not use MOVE BY NAME.
- You cannot use the $REC-SEGMENT function in any statement other than the SET statement and not as the source of data in the SET statement.
- This function is not accessible outside the scope of a FOR statement.
- If you are updating an ESDS data set, you cannot actually change the length of an existing record. However, you can still specify the $REC-SEGMENT function to change the apparent length of the record. If the updated record is shorter than the original record, the record is padded with binary zeros. If the record is longer than the original record, a run-time error occurs.

  When updating variable-segment records, you must determine which variable segment is included in the input record. If the record is changed to require a different variable segment, you must set the $REC-SEGMENT function to write the record correctly. Usually, the fixed segment of the record contains a field (frequently known as record type) that indicates which type of variable segment the record has. You can test the value of that field to determine what type of record was read.

  When you write variable-segment records without setting the $REC-SEGMENT function, the following defaults apply:
- **FOR NEW** Uses the length of the first and largest variable segment.
- **FOR EACH/FIRST** Uses the length of the record that was just read.

Examples

The following statement causes CA Ideal to write a record in the VSAM dataview named MASTER-SUMMARY using the variable segment named TYPE1. TYPE1 is the name of a level-2 group field defined in the MASTER-SUMMARY dataview. Another variable segment, named TYPE2, redefines the TYPE1 group field.

```
    FOR NEW MASTER-SUMMARY
    …
    .
    .
    SET $REC-SEGMENT(MASTER-SUMMARY) = 'TYPE1'
ENDFOR
```

The next example shows how the $REC-SEGMENT function changes the record type in a FOR EACH statement.

```
    FOR EACH MASTER-SUMMARY
   IF REC-TYPE = 1
     THEN
       SET REC-TYPE = 2
       SET $REC-SEGMENT(MASTER-SUMMARY) = 'TYPE2'
       statements             statements changing the record
    ENDIF
ENDFOR
```

The field named REC-TYPE is tested first to determine which type of record was read. Then the record is changed, and finally, the $REC-SEGMENT function changes the record type to the one that includes the variable segment named TYPE2. When the record is written at the ENDFOR statement, the record type is changed from 1 to 2, and the variable segment TYPE2 is written.

$REMAINDER Function

$REMAINDER returns the remainder after one numeric expression is divided by another. Mathematically, the remainder is defined as

```
m - ([m|n] * n)
```

where [ ] means “the integer portion of.”

This function has the following format:

```
$REMAINDER(m,DIV=n)
```

m

The dividend. This must be a numeric expression.

DIV=n

The divisor. This must be a numeric expression.

Examples

```
SET I = 100
SET J = 2567
SET K = $REMAINDER (J,DIV=I) : remainder is 67

SET I = 100
SET J = -2567
SET K = $REMAINDER (J,DIV=I) : remainder is -67
```

$RETURNCODE Function

$RETURN-CODE is a pseudo function that returns the highest return code caused by a system error or set by a CA Ideal program during an online run or batch session.

This function has the following format:

```
$RETURN-CODE
```

or

```
$RC
```

When a system error is encountered, $RETURN-CODE is set to a value that depends on the severity level. Each system message has a message level with an associated return code. The program can also explicitly set the return code to any value. For commands other than RUN (which can set any value for $RC), the following table shows how $RETURN-CODE values are associated with warning and error messages.

Message Level	**Return Code**
I - Information	0
A - Advisory	4
W - Warning	4
E - Error	8
F - Fatal Error	12
C - Conditional	16
D - Disaster	16
T - Terminal	16

$RETURN-CODE is set to the returned value only if the current value of the function is lower than the returned value.

In a PDL procedure, you can use $RETURN-CODE as a sending field in any context where numeric fields can be used (for example, in numeric expressions and conditional expressions). When used in a SET statement, $RETURN-CODE can be a sending or a receiving field. The SET statement unconditionally changes the value of $RETURN- CODE. In a MOVE statement, $RETURN-CODE can only be a sending field or part of a sending numeric expression.

Execution of a default error procedure sets the value of $RETURN-CODE to 12 when the value is less than 12. Otherwise, the value remains unchanged. A user defined error procedure is not called on a system error, so it changes the value of $RETURN-CODE only if explicitly coded to do so.

At the end of a run, the message 'RUN completed, RC=nn' appears. The RC=nn is the value of the return code at the end of the run.

In a batch jobstream, you can use $RETURN-CODE with CA Ideal's IF, ELSE, and ENDIF commands to conditionally execute other CA Ideal commands. You can also use return codes to ensure that programs in a batch environment do not run unless the previous program executes successfully. The return code value is passed to the operating system at the end of a CA Ideal batch session. For more information, see the *Messages and Codes Guide*.

Return code values set before a run by another RUN or a CA Ideal command are retained at the start of a new RUN when SET RUN $RC KEEP is in effect. SET RUN $RC ZERO resets the return code to zero at the start of a run.

Note: The LIST ERROR statement automatically displays the value of $RETURN-CODE.

Example

In the following example, the $RETURN-CODE is set to different codes, depending on which WHEN statement qualifies. When the $ERROR-CLASS is DVW, the $RETURN-CODE is set to 1660 to indicate a DVW error. The LIST ERROR codes, LIST ERROR, and QUIT RUN statements are performed.

When the ERROR-CLASS is NUM, the $RETURN-CODE is set to 8. The PROCESS NEXT MAIN-LOOP statement returns the control to the main loop. The $RETURN-CODE prints as a warning at the end of the run unless higher $RETURN-CODEs are incurred.

When the $ERROR-CLASS is any other value, the $RETURN-CODE is set to 12. This indicates that an unexpected error was found and the procedure does a LIST ERROR and QUIT RUN.

```
<<ERROR>> PROCEDURE
    SELECT $ERROR-CLASS
    WHEN 'DVW'
        SET $RETURN-CODE = 1660:Use a high number which
                               :you might choose
        LIST $ERROR-DVW-DBID   :to indicate a dvw error
        LIST $ERROR-DVW-STATUS
        LIST $ERROR-DVW-INTERNAL-STATUS
    WHEN 'NUM'
        SET $RETURN-CODE = 8
        PROCESS NEXT MAIN-LOOP
     WHEN OTHER
        SET $RETURN-CODE = 12
    ENDSEL
    LIST ERROR
    QUIT RUN
   ENDPROC
```

$ROUND Function

$ROUND returns a value that is the input value rounded by the specified or implied factor.

This function has the following format:

```
                  {FACTOR=f }
$ROUND (expression,{ATTR=id  } )
```

expression

Defines a numeric expression.

FACTOR=f

Indicates that the value is obtained by rounding the numeric expression to the nearest value according to a rounding factor.

f

A rounding factor. This factor is any numeric identifier or numeric literal with a positive value. If the factor has a negative value, the sign is ignored and assumed to be positive. If the factor is 0, no rounding is performed.

For example, if the rounding factor is 1, the numeric expression is rounded to the nearest integer (with the rule that a value with a fractional part equal to or greater than .5 rounds to the next higher number). A rounding factor of .1 rounds the numeric expression to the nearest tenth. A rounding factor of 25 rounds the expression to the nearest multiple of 25, and so on.

ATTR=id

Indicates that the value is obtained by rounding to the nearest unit based on the attributes of the specified identifier. For example, if the specified identifier has 2 decimal places, that is equivalent to a FACTOR of .01.

id

The identifier of a numeric field whose attributes designate the rounding factor. If the specified field is subscripted, the subscript is not required as part of the identifier.

Examples

```
SET I = $ROUND (257.6,FACTOR=1)  : result is 258
SET I = $ROUND (257.2,FACTOR=1)  : result is 257
SET I = $ROUND (257.2,FACTOR=0)  : result is 257.2
SET I = $ROUND (257,FACTOR=50)   : result is 250
SET I = $ROUND (285,FACTOR=50)   : result is 300
SET I = $ROUND (-285,FACTOR=50)  : result is -300
SET I = $ROUND (2.378,FACTOR=.01): result is 2.38
SET I = $ROUND (2.372,FACTOR=.01): result is 2.37
SET I = $ROUND (2.378,ATTR=K)    : assuming K has 2
                                 : decimal places,
                                 : result is 2.38
```

$RRN Function

The $RRN function is used with VSAM RRDS files to access a specific record or range of records. This function returns the current relative record number at each iteration of the FOR construct. You can use the $RRN function in the WHERE clause of a FOR statement, as the source operand in SET statements, and in expressions.

This function has the following format:

```
$RRN[(dataview-name)]
```

dataview-name

The name of a VSAM dataview that defines an RRDS file. The dataview name is required for this function, except when the function is used in a WHERE clause. You cannot specify the dataview name for this function in the WHERE clause.

- You cannot set the value of the $RRN function , therefore, you cannot use the $RRN function as the target of a SET or MOVE statement.
- The $RRN function is accessible only when the dataview fields are accessible. You cannot use this function before the execution of a FOR statement.
- When adding records to an RRDS file, the WHERE clause is required and the $RRN function is used as the left operand of the WHERE clause. The operator must be equal to (=) and the right operand can be an integer number, a numeric expression, or a numeric data item or field.
- The first record of an RRDS file has a relative record number of 1.

Examples

In the following example, the $RRN function is used in the WHERE clause in a FOR NEW statement to add records in consecutive slots in an RRDS file named PAYROLL. To find the last record in the existing file, a FOR FIRST construct is used with an ORDERED BY DESCENDING clause. The working data field NEXT-REC is set by adding 1 to the number the $RRN function returns.

```
FOR FIRST PAYROLL
   ORDERED BY DESCENDING
        SET NEXT-REC = $RRN(PAYROLL) + 1
WHEN NONE
        SET NEXT-REC = 1

    ENDFOR
FOR NEW PAYROLL
    WHERE $RRN = NEXT-REC
        statements
        SET NEXT-REC = $RRN(PAYROLL) + 1
ENDFOR
```

In the next example, the $RRN function finds a specific record in the PAYROLL file.

```
FOR FIRST PAYROLL
    WHERE $RRN = 6
        statements
ENDFOR
```

$SPACES Function

This function returns a space or blank.

This function has the following format:

```
$SPACES
```

- You can use $SPACES as a source field in a MOVE or SET statement. It assumes the same length as its associated target.
- You can use $SPACES in a conditional expression as the object of a comparison. It assumes the same length as its associated comparand.
- You can use $SPACES as an implied comparand in a SELECT construct. $SPACES assumes the length of its associated source comparand.
- $SPACES has a length of 1 when used as the argument of $STRING or in a LIST statement.

Example

Assume that A, B, and C are 16-character alphanumeric fields.

```
SET A = $HIGH       : returns 16 X'FF'
    SET B = $LOW        : returns 16 X'00'
    SET C = $SPACES     : returns 16 X'40'
```

$SQL Functions (SQL Access Only)

The $SQL functions return information about the last SQL statement executed in the current run unit.

Last SQL Statement

The following function identifies the last SQL statement executed. It returns the statement number and the program name (in the form *sys-id.program-name*).

Function	Data Type
$SQL-LAST-STMT	V36

Last DBMS Accessed

The following function is used to display the last type of database accessed. It returns DBSQL or DB2.

Function	Data Type
$SQL-DBMS	X8

SQLCA Data

Each of the following $SQL functions returns the value from an SQLCA field for the last SQL statement executed. The data type of each function is the data type of the associated SQLCA field. The functions, their associated fields, and their data types follow.

For Any SQL Access

The following functions display SQLCA data for both DB2 and CA Datacom/DB SQL access.

Function	Synonym	SQLCA Field	Data Type
$SQLCAID	$SQLCA-EYE-CATCH	SQLCAID	X8
$SQLCABC	$SQLCA-LEN	SQLCABC	NB9
$SQLCODE		SQLCODE	NB9
$SQLERRM	$SQLCA-ERROR-INFO	SQLERRM	V80
$SQLERRP	$SQLCA-ERROR-PGM	SQLERRP	X8
$SQLWARN0		SQLWARN0	X1
$SQLWARN1		SQLWARN1	X1
$SQLWARN2		SQLWARN2	X1
$SQLWARN3		SQLWARN3	X1
$SQLWARN4		SQLWARN4	X1
$SQLWARN5		SQLWARN5	X1
$SQLWARN6		SQLWARN6	X1
$SQLWARN7		SQLWARN7	X1
$SQLSTATE		SQLSTATE	X5

SQLCA Data for DB2

The following functions display SQLCA data for DB2 SQL access only. (If the last SQL access was for CA Datacom/DB, these functions return N/A.)

Function	SQLCA Field	Data Type
$SQLERRD1	SQLERRD1	NB9
$SQLERRD2	SQLERRD2	NB9
$SQLERRD3	SQLERRD3	NB9
$SQLERRD4	SQLERRD4	NB9
$SQLERRD5	SQLERRD5	NB9
$SQLERRD6	SQLERRD6	NB9
$SQLEXT	SQLEXT	X8
$SQLWARN8	SQLWARN8	X1
$SQLWARN9	SQLWARN9	X1
$SQLWARN10	SQLWARN10	X1

SQLCA Data for DATACOM SQL

The following functions display SQLCA data for CA Datacom SQL access only. If the last database accessed was DB2, these functions return N/A.

Function	SQLCA Field	Data Type
$SQLCA-DB-VRS	SQLCA-DB-VRS	X2
$SQLCA-DB-RLS	SQLCA-DB-RLS	X2
$SQLCA-LUWID	SQLCA-LUWID	X8
$SQLCA-DSFCODE	SQLCA-DSFCODE	X4
$SQLCA-INFCODE	SQLCA-INFCODE	NB9
$SQLCA-DBCODE-EXT	SQLCA-DBCODE-EXT	X2
$SQLCA-DBCODE-INT	SQLCA-DBCODE-INT	NB4
$SQLCA-PGM-NAME	SQLCA-PGM-NAME	X8
$SQLCA-AUTHID	SQLCA-AUTHID	X18
$SQLCA-PLAN-NAME	SQLCA-PLAN-NAME	X18

$SQRT Function

$SQRT returns the square root of a numeric expression.

This function has the following format:

```
$SQRT (numeric-expression)
```

If the numeric expression is negative, an error condition is raised.

Example

```
    SET I = $SQRT(J)         :If J is 36, I is 6.
```

$STRING Function

$STRING returns an alphanumeric value obtained from the series of parameters by concatenating the values obtained for each parameter.

This function has the following format:

```
$STRING (parm-1,...,parm-n)
```

parm-1,...,parm-n

Defines the items to concatenate (after conversion to alphanumeric expressions, if necessary). An expression must be one of those described follows. If any items contain null values, the concatenated string returns null values. The actions of $STRING vary with the parameter type as follows:

- **alphanumeric-**field or literal No change.
- **flag-** Converted to a one-character value of T or F.
- **group-** Converted to the concatenation of the result of $STRING on each of the fields in the group. If the group contains an OCCURs, all occurrences are concatenated. Redefined items in the group are ignored.
-
- **numeric field-**Converted to alphanumeric representation as follows: if the number is negative, a minus sign is generated, otherwise, no sign representation appears. The integer portion of the number is represented with the display form of the numerals, with leading zeros according to the attributes of the field. The leading zeros, including the units position, are converted to leading blanks.

  Note: You can use the $EDIT function to edit a given field or literal according to a specified pattern. For more information, see the $EDIT Function section.

  If, in this interim result, leading blanks and a minus sign appears, the minus sign is floated to the position adjacent to the first significant digit. If the field has one or more decimal positions, a decimal point is generated, followed by the decimal part with trailing zeros according to the attributes of the field. A byte must be allocated in the field size for an addition or minus sign or a decimal point.

- **date field-**Converted to the default SCF date pattern (for example, mm/dd/yy). The SET COMMAND SESSION OPTIONS command displays the format.
- **numeric literal-**Each numeric digit is converted to the corresponding alphanumeric numeral; a plus sign is converted to a blank; a minus sign is retained.
- **alphanumeric function-** You can nest alphanumeric functions used as parameters of $STRING to three levels. $SPACE, $HIGH, and $LOW are each assumed to be one character long when used as parameters of $STRING.

Note: You must specify each occurrence of fields that are defined as repeating fields (defined with OCCURs) with a subscript.

Examples

```
$STRING ('ABC', 'DEF')            : result is 'ABCDEF'
```

Assume G is a group that contains I, J, K.

Assume I is an unsigned numeric field with three digits and two decimal places.

Assume J is a signed numeric field with six digits.

Assume K is an alphanumeric elementary five-character item.

```
SET I = 34.5
SET J = -123
SET K = 'HELLO'
```

```
$STRING (I)      : result is '  34.50'
$STRING (J)      : result is '   -123'
$STRING (K)      : result is 'HELLO'
$STRING (I,J)    : result is '  34.50   -123'
$STRING (J,K)    : result is '   -123HELLO'
$STRING (G)      : result is '  34.50   -123HELLO'
$STRING (I,J,K)  : result is '  34.50   -123HELLO'
```

$STRING(G) and $STRING(I,J,K) are equivalent.

```
$STRING ('*' $SUBSTR('ABCDE' START=2,LENGTH=3) $SPACE K '*')
                 : result is '*BCD HELLO*'
```

$SUBSCRIPTPOSITION Function

This function returns the position of the subscript (1, 2, or 3). In error processing, use the $ERROR-SUBSCRIPT function to determine the position of the subscript in error.

$SUBSTR Function

$SUBSTR returns an alphanumeric expression that is part (or all) of another alphanumeric expression.

This function has the following format:

```
$SUBSTR (alpha-expression [,START=start][,LENGTH=len])
```

alpha-expression

Defines an alphanumeric expression.

start

Defines a numeric expression to be extracted whose value is the starting position of the alphanumeric expression. It cannot be a nullable expression. If the start value exceeds the length of the alphanumeric expression, then a null value is returned (a value of length 0). If assigned to an alphanumeric field, the field attains a value of all spaces according to the usual padding rules for SET or MOVE. The default start position is 1. If the start value is less than 1, 1 is assumed.

len

Identifies the number of characters to extract from the alphanumeric expression beginning at the start. You must specify the length as a numeric expression. It cannot be a nullable expression. If you omit len, all characters from the start to the end of the alphanumeric expression are extracted. If the value of len exceeds the remaining length of the alphanumeric expression, then the remaining length is used. If the *len* value is less than 1, a null value is returned (see START).

The start and length parameters are optional. You can use them independently of each other. You can use them in either order.

You must define a numeric field specified for the start or length parameter with integer digits only. When you specify any other kind of numeric expression, if the value is not a whole number, the integer portion is used.

Examples

Assume that A is a 16-character alphanumeric and B is a five-character alphanumeric.

```
SET A = 'THESE THREE WORDS'
SET B = $SUBSTR (A,START=7,LENGTH=5)  :result B='THREE'
```

$SYSTEM Function

The $SYSTEM function returns the three-character ID of the system containing the current program. It is always available, unlike the corresponding $ERROR-SYSTEM variable.

This function has the following format:

```
$SYSTEM
```

- You can use this function for security testing or for accounting purposes.

$TERMINALID Function

The $TERMINAL-ID function returns a four-character terminal identification in the form of an alphanumeric value.

This function has the following format:

```
$TERMINAL-ID
```

- You can use this function for security testing or for accounting purposes.

$TIME Function

$TIME returns an alphanumeric value with the specified or current time in the specified format.

This function has the following format:

```
$TIME ('time-specification'[,TIME=time])
```

'time-specification'

A sequence of characters (maximum 30) that represent components and notation of the time, as follows:

```
Component                                                   Example (Assuming
Notation    Meaning                                          (:03:08 am)

H           Hour without leading zeros
              (24-hour clock)                               9
HH          Hour with leading zero                           09
PP          Hour with leading zero, (12-hour
              clock with AM/PM indication)                   09 AM
P           Same but without leading zero                    9 AM
MM          Minutes with leading zero                        03
SS          Seconds with leading zero                        08
```

Any character in the format, except uppercase alphabetics, remains unchanged. You must explicitly specify spacing and punctuation.

time

A six-character alphanumeric literal in the format of 'HHMMSS' or a six-character alphanumeric field with a value in that format. If you do not specify the TIME operand, the current time is used.

- The TIME operand must contain six digits in alphanumeric format.
- The TIME operand must be in the format 'HHMMSS', where 'HHMMSS' can be from 000000 to 240000. In this range, SS can be from 00 to 59, MM can be from 00 to 59, and HH can be from 00 to 23. The time is based on a 24-hour clock that starts on 000000 (midnight), continues through 235959, and then resets to 000000. 240000 is also acceptable for midnight.
- If you omit the TIME operand, a call is generated to the operating system.

Examples

Assume that it is 9:03:08 am and FIELD contains 121336

$TIME Function Invocation	Result
$TIME ('HH:MM:SS')	09:03:08

$TIME Function Invocation	Result
$TIME ('H:MM')	9:03
$TIME ('PP:MM')	09:03 AM
$TIME ('HH-MM-SS',TIME=FIELD)	12-13-36

$TODAY Function

$TODAY is a numeric function that returns the CA Ideal internal integer date for the current date (that is, the date at runtime).

This function has the following format:

```
$TODAY
```

$TRANSACTIONID Function

This function returns the four-character ID of the transaction that accesses CA Ideal.

This function has the following format:

```
$TRANSACTION-ID
```

$TRANSLATE Function

$TRANSLATE returns an alphanumeric value that consists of the argument value with all occurrences of specified characters translated to other specified characters.

This function has the following format:

```
$TRANSLATE (alphanumeric-expression, FROM=from-characters,      TO=to-characters)
```

alphanumeric-expression

The alphanumeric expression to convert.

FROM= from-characters

The source characters for translation. This can be an alphanumeric literal or the name of an alphanumeric field or an alpha-group. Each of the from-characters must be unique. The number of from-characters and to-characters are usually in a one-to-one correspondence. If not, the length of the shorter character string is taken and the remainder of the longer string ignored.

If the lengths are not equal, a warning is issued during compilation.

If from-characters is a literal and there is a duplication of one of the characters, a compile error is issued. If from-characters is an identifier and there is a repetition of one of the characters, then the last occurrence is assumed.

TO=to-characters

Specifies the target characters for translation. This can be an alphanumeric literal, alphanumeric field, or alpha-group.

Note: The reserved word parameters TO and FROM can appear in either order.

Examples

Assume that A and B are 16-character alphanumeric fields.

```
SET A = 'ABC.DEF,GHI;JKL.'
SET B = $TRANSLATE (A, TO='/$', FROM='.,')
: convert periods to slashes and commas to dollar signs;
  result in B
: is 'ABC/DEF$GHI;JKL/'.  The result could have been
  assigned back
: to A.
```

The following example illustrates how you can use $TRANSLATE to ensure that input that was entered in mixed case can process as all uppercase.

```
SET UPPER FIELD = $TRANSLATE(SOURCE-FIELD,
FROM='abcdefghijklmnopqrstuvwxyz',
TO='ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

You could also use the $UPCASE function to do the same translation, if your site translation table, PMSTRUC, matches your requirements. $TRANSLATE is useful for situations where the translation to upper case also involves such things as removing accents.

In the following example, when you use the FROM character more than once, the corresponding rightmost TO value is used in the translation.

```
SET F1 = "111"
SET T1 = "ABC"
SET X1 = "1001"
SET X2 = $TRANSLATE(X1,FROM=F1,TO=T1)
```

Therefore, the value of X2 is C00C.

$TRIM Function

$TRIM is an alphanumeric function that returns the string resulting from the removal of leading or trailing characters from an alphanumeric expression.

This function has the following format:

```
                       [, RIGHT= {'a'     }]
$TRIM(alpha-expression [, LEFT=  {a-field }].)
```

alpha-expression

Defines an alphanumeric expression.

RIGHT=|LEFT=

The RIGHT= clause trims from the right any trailing characters with the specified value. The LEFT= clause trims from the left any leading characters with the specified value. You can specify either clause or both. If neither clause is specified, the default is RIGHT=' ' (trim trailing blanks).

'a'

Specifies an alphanumeric literal indicating the character to trim.

a-field

Specifies an one-byte alphanumeric field containing the character to trim.

- The function does not change the alphanumeric expression, but uses it to build the expression that is returned.
- If the specified character is not found, the expression is returned unchanged.

Examples

The following example trims leading and trailing blanks from a dataview field and processes the resulting field if it is less than or equal to eight characters long.

```
SET VAR-FLD = $TRIM(DVW.FIELD,LEFT=' ',RIGHT=' ')
    IF $LENGTH(VAR-FLD) LE 8
   DO MOVE-ROUTINE
ENDIF
```

Moving the expression $TRIM returns to a field can produce different results for fixed length fields and variable length fields. For example, the fields used are defined as follows:

```
Field FIX-FLD:  Type X, length 10, initial value 'ABC
Field VAR-FLD:  Type V, length 10, initial value 'ABC'
```

The following statement sets FIX-FLD to 'ABCbbbbbbb', where b is a blank.

```
SET FIX-FLD = $TRIM(FIX-FLD, RIGHT=' ')
```

The function trims the trailing blanks from the original expression, but because the target field is fixed length, ten characters long, the trailing blanks are added back when the string is moved.

On the other hand, the following statement sets VAR-FLD to ABC because you can trim the variable-length target field to three characters.

```
SET VAR-FLD = $TRIM(FIX-FLD,RIGHT=' ')
```

See the $PAD function.