COMPUTER
CORPORATION
OF AMERICA

Model 204®

User Language Manual
Parts III — VI

Version 5 Release 1

UL251A

Computer Corporation of America

Corporate Headquarters:		*CCA International:*	
500 Old Connecticut Path Framingham, MA 01701 USA		First Floor, Edinburgh House 43-51 Windsor Road Slough, Berkshire SL1 2EE England	
Phone:	(508) 270-6666	Phone:	+44-1753-472800
Fax:	(508) 270-6688	Fax:	+44-1753-472888

Model 204 User Language Manual: Parts III—VI
Document Number UL251A
March 27, 2002

Contents

Model 204®

User Language Manual
Parts III — VI

Preface

Preface

The Model 204 *User Language Manual* describes User Language, the fourth-generation programming language for Model 204. This manual is divided into two volumes.

Audience

Consult this manual if your need to execute standard Model 204 functions, including:

- Retrieve stored information.
- Display and print retrieved information in a desired format.
- Perform arithmetic or conditional operations with stored information.
- Store new information.

Introducing Model 204 electronic documentation

Model 204 documentation includes several other manuals to which you might want to refer. The CD-ROM, titled *Model 204 Documentation*, contains the most recently released documentation for Model 204.

The document files are in Portable Document Format; each has a PDF file extension. You can view, navigate, and print the individual manuals, and you can search the entire document set using Adobe™ Acrobat Reader™ with Search software, which is also provided on the CD-ROM. Either view the manuals directly from the CD-ROM, or download the files to a network server.

In the PDF directory, open the README.TXT file on the Windows Notepad. This file includes instructions to download a copy of the Acrobat Reader with Search and to open the *Model 204 Documentation Library Catalog*.

Note: You may access the documentation online or print out copies, as needed. However, consistent with the terms of your license agreement, you may not copy or distribute the CD-ROM, or distribute hard-copies to third parties.

Contacting CCA Customer Support

If you need assistance with this product beyond the provided online help and documentation, and you have licensed this product directly from CCA, either call CCA Customer Support at 1-800-755-4222, or access the Customer Support section of the CCA Web site. The Web address is:

`http://www.cca-int.com`

If you have not licensed this product directly from CCA, please consult your vendor.

Notation conventions

This manual uses the following standard notation conventions in statement syntax and examples:

Convention	Description
`TABLE`	Uppercase represents a keyword that you must enter exactly as shown.
`TABLE` *`tablename`*	In text, italics are used for variables and for emphasis. In examples, italics denote a variable value that you must supply. In this example, you must supply a value for *tablename*.
`READ [SCREEN]`	Square brackets ([ ]) enclose an optional argument or portion of an argument. In this case, specify READ or READ SCREEN.
`UNIQUE | PRIMARY KEY`	A vertical bar (|) separates alternative options. In this example, specify either UNIQUE or PRIMARY KEY.
`TRUST |` <u>`NOTRUST`</u>	Underlining indicates the default. In this example, NOTRUST is the default.
`IS {NOT | LIKE}`	Braces ({ }) indicate that one of the enclosed alternatives is required. In this example, you must specify either IS NOT or IS LIKE.
`item …`	An ellipsis (…) indicates that you can repeat the preceding item.
`item ,…`	An ellipsis preceded by a comma indicates that a comma is required to separate repeated items.
`All other symbols`	In syntax, all other symbols (such as parentheses) are literal syntactic elements and must appear as shown.
`nested-key` `::=` *`column_name`*	A double colon followed by an equal sign indicates an equivalence. In this case, *nested-key* is equivalent to *column_name*.
`Enter your account:` `sales11`	In examples that include both system-supplied and user-entered text, or system prompts and user commands, boldface indicates what you enter. In this example, the system prompts for an account and the user enters **sales11**.

Convention	Description
File > Save As	A right angle bracket (>) identifies the sequence of actions that you perform to select a command from a pulldown menu. In this example, select the Save As command from the File menu.

Part III
Advanced Features and Considerations

Part III covers features and considerations which allow you to take further advantage of the flexibility of the Model 204 file structure, pass information between requests, and manage lengthy requests effectively. Included are chapters on:

- Multiply occurring fields
- Global variables, lists, and found sets
- Writing large requests

19

Operations on Multiply Occurring Fields

In this chapter

Overview

A single field name that has different values can be stored repeatedly in a record. For example, the field CHILD in the record shown below is an example of a ***multiply occurring field***:

```
FATHER = JOHN DOE
CHILD = ELIZABETH
CHILD = ROBERT
```

Any field name *except* for the following types of fields can be present more than once in a record:

- A NUMERIC RANGE field
- A sort key
- A hash key

Special processing for multiply occurring fields

Certain User Language statements operate differently on multiply occurring fields than on singly occurring fields. User Language provides special statements and forms of statements that can be used with multiply occurring fields—specifically introducing the EACH modifier and subscripts. User Language also provides for the use of subscripted field references for accessing a particular occurrence of a multiply occurring field.

FIND statement

Retrieval

Example 1 Assume that we are using this sample record:

```
FATHER = JOHN DOE
CHILD = ELIZABETH
CHILD = ROBERT
```

You can use either of these two statements to retrieve the record:

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD = ELIZABETH
           END FIND
```

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD = ROBERT
           END FIND
```

Example 2 The record is *not* retrieved by either of these statements:

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD = NOT ELIZABETH
           END FIND
```

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD = NOT ROBERT
           END FIND
```

Multi-condition range retrievals

You should pay special attention to the effect of multiply occurring fields on multi-condition range retrievals. Undesirable results can occur if a range search is specified for a multiply occurring field. For example:

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD IS AFTER KEN AND BEFORE PAUL
           END FIND
```

retrieves the sample record even though neither child’s name in the sample is between KEN and PAUL. Model 204 evaluates each condition of the FIND separately, then combines the results of each evaluation to build the set of records satisfying all conditions. The BETWEEN operator behaves exactly like any other multi-condition range retrieval when used on a multiply occurring field.

If a range search must be performed on a multiply occurring field, use the IN RANGE clause of the FIND statement. Refer to “NUMERIC RANGE and ORDERED NUMERIC attributes” on page 4-5 for more information about the IN RANGE clause. Better code for the previous example would be:

```
FIND.RECS: FIND ALL RECORDS FOR WHICH
              CHILD IS ALPHA IN RANGE
              AFTER KEN AND BEFORE PAUL
           END FIND
```

Use of subscripts

You cannot use subscripted references with the FIND statement. Refer to “Subscripts” on page 19-21 for more information about using subscripts with multiply occurring fields.

NOTE statement

Only first occurrence is noted

The NOTE statement notes only the first occurrence of a multiply occurring field. For example, only the first occurrence of the field (CHILD = ELIZABETH) are noted by:

```
KEEP.CHILD: NOTE CHILD
```

Use of subscripts

In order to note a particular occurrence of a multiply occurring field, the field name must be subscripted. Refer to the discussion of subscripts on “Subscripts” on page 19-21 for detailed information on subscripted field names and usage.

PRINT and PRINT *n* statements

PRINT statement output format

The PRINT statement prints only the first occurrence of a field in a record.

If there is more than one value of a field in a record, the special modifier, EACH, can be used in a PRINT statement to print out all the values on a single line, with a single space between values.

Example 1

```
PRINT EACH INCIDENT
```

yields:

```
T1 T2 T1 T3 T2 T1
```

Example 2

If the field is given a column position, as in:

```
PRINT FULLNAME WITH EACH INCIDENT AT COLUMN 18
```

values are printed one to a line and positioned at the column specified.

```
ABBOTT, GAIL H    T1                       -
                  T3                       -
                  T1                       -
                  T3                       -
                  T2                       -
                  T1
```

Example 3

A field cited in a PRINT statement after a multiply occurring field is printed on the same line as the last value of the multiply occurring field. If you were to change the PRINT statement in the sample request as here:

```
PRINT FULLNAME WITH EACH INCIDENT AT COLUMN18 -
      WITH POLICY NO AT COLUMN 23 -
       WITH STATE AT COLUMN 32
```

this output results:

```
ABBOTT, GAIL H T1                           -
T3                                          -
T1                                          -
T3                                          -
T2                                          -
T1 100340 CALIFORNIA
```

Use of subscripts

See “PRINT statement” on page 19-25 for a description of using PRINT with subscripts.

PRINT *n* statement

Long fields, such as abstracts, evaluations, or statements of purpose, can be stored as a multiply occurring field. Each occurrence can contain as many as 255 characters.

Syntax

You can use this PRINT statement to print such a field as a paragraph:

```
PRINT n fieldname
```

where *n* must be a number less than 32,768.

Output format

The PRINT *n* statement prints up to *n* lines of text, composed of all of the occurrences of the field concatenated in order. Nothing is inserted. At most, *n* lines are printed; any extra lines are ignored.

Ordinarily, a PRINT statement that produces more than one line inserts a hyphen in the continuation column of the output device. Instead of using the continuation column, this form of PRINT attempts to end each line with a complete word delimited by spaces. If there is insufficient space to fit the last word on a line, the word is hyphenated arbitrarily and continued on the next line.

The AT COLUMN and TO COLUMN clauses can be used to adjust the output to a narrower column. When used together with PRINT *n*, text is broken at word boundaries to fit within the column. The AT clause does not affect the first line, and the TO clause does not affect the last line (unless the line limit *n* is exceeded). This allows indenting.

The AT or TO column options cannot accept negative numbers or numbers greater than 32767 for the column.

Procedures containing PRINT statements with negative numbers or numbers greater than 32767 fail at compile time with the following counting error message:

```
M204.0263: AT/TO MUST BE BETWEEN 1 AND 32767
```

Example

This example illustrates the use of the PRINT *n* statement.

```
BEGIN
            PRINT ’1234567890123456’
            STORE RECORD
               TEXT = ’NOW IS THE T’
               TEXT = ’IME FOR’
               TEXT = ’ ALL GOOD MEN TO’
               ITEM = 1
            END STORE
FD.REC:     FIND ALL RECORDS FOR WHICH
               ITEM = 1
            END FIND
            FOR 1 RECORD IN FD.REC
```

```
PRINT.TEXT:    PRINT ’’ AT COLUMN 5 WITH 3 TEXT -
                  AT COLUMN 3 TO 16
            END FOR
END
```

The output produced is:

```
1234567890123456
    NOW IS THE
  TIME FOR ALL
  GOOD MEN TO
```

The PRINT.TEXT statement concatenates a null value (two single quotes with no space between them) with the PRINT n form to indent the first line of text by using the location of the null value (column 5).

Note that the printing stops short of column 16 to avoid truncating TIME and GOOD.

Use with subscripts

You cannot use subscripted references with the PRINT *n* statement. Refer to “Subscripts” on page 19-21 for more information about using subscripts with multiply occurring fields.

ADD, CHANGE, and DELETE statements

The ADD and CHANGE statements, and both forms of the DELETE statement, are supported in remote file and scattered group contexts.

ADD statement

The ADD statement places new occurrences of a field after existing occurrences. For example, if new children are added to the sample record, the additions are placed last. Thus:

```
ADD CHILD = SARAH
ADD CHILD = PATRICK
```

results in the record:

```
FATHER = JOHN DOE
CHILD = ELIZABETH
CHILD = ROBERT
CHILD = SARAH
CHILD = PATRICK
```

The INSERT statement

If the order of occurrence is important, the INSERT statement can be used to add new occurrences. See the discussion on “INSERT statement” on page 19-24.

CHANGE statement

The CHANGE statement alters only the first occurrence of a field in a record.

Syntax

You can use this form of the CHANGE statement if there is more than one occurrence:

```
CHANGE fieldname = value1 TO value2
```

This form, like the CHANGE fieldname statement, can be used only inside a FOR EACH RECORD loop. It deletes the first occurrence of the pair *fieldname = value1* from a record, and adds the pair *fieldname = value2* to the record. The new value is added either in the position occupied by the original value or at the end of the record, depending upon the update attribute specified for the field by the file manager. See the discussion in “UPDATE field attribute” on page 19-19.

If the specified field, *fieldname = value1*, does not appear in the record, *fieldname = value2* is simply added to the record. If the specified *fieldname = value1* pair appears more than once in the record, only the first occurrence of it is deleted. The pair, *fieldname = value2*, is added just once. Occurrences of the field name that have other values are not altered by the statement.

DELETE statement

The DELETE statement deletes only the first occurrence of the field in the record by default.

Two forms of DELETE statement

For multiply occurring fields, two forms of the DELETE statement are provided; however only for use inside a FOR EACH RECORD loop:

To delete…	Syntax	Usage
A particular occurrence	DELETE *fieldname = value*	To delete the first occurrence of the pair, *fieldname = value*, from a record. Occurrences of the field that have other values are not removed. If the field with the specified value: • Occurs more than once in the record, only the first occurrence is deleted. • Cannot be found, no deletion occurs.
Every occurrence of the field in the record	DELETE EACH *fieldname*	To delete all occurrences of the specified field name. The field to be deleted cannot have the INVISIBLE attribute.

Use with the FOR EACH OCCURRENCE statement

See “Deleting occurrences” on page 19-16 for a discussion of deleting occurrences using the FOR EACH OCCURRENCE OF (FEO) statement.

Use with subscripts

See “DELETE statement” on page 19-25 for a discussion of using the DELETE statement with subscripts.

SORT RECORDS statement

When a multiply occurring field is chosen as a sort key, each record in the set being sorted is processed once using the first occurrence of the field as the key.

EACH modifier with one sort field

If the EACH modifier is present in the SORT statement and if there are *n* occurrences of the field in the record, similar records are created in the sorted copy of the original set.

Example The following request:

```
SORT RECORDS IN FIND.RECS BY EACH KEY
```

generates three temporary records for the single permanent record in which the field named KEY occurs three times:

```
KEY = COMPUTER
KEY = CORPORATION
KEY = AMERICA
```

The records generated are:

```
1 KEY = COMPUTER        2 KEY = CORPORATION
  KEY = CORPORATION       KEY = AMERICA
  KEY = AMERICA           KEY = COMPUTER

3 KEY = AMERICA
  KEY = COMPUTER
  KEY = CORPORATION
```

These correspond to the n cyclic permutations of the set of field occurrences and, in this example, are sorted into the order 3, 1, 2 (if no other option is selected). Statements that refer to the sorted set, such as PRINT, PRINT EACH, NOTE, and PRINT ALL INFORMATION, reflects the permutation. No record is generated if n = 0.

EACH modifier with several key fields

When the EACH option is selected for several keys, *n* occurrences of one key and m of another produce differently permuted records. If either *n* or *m* equal 0, no records are generated.

Example This request:

```
BEGIN
FIND.RECS: IN CLIENTS FIND ALL RECORDS FOR WHICH
              INCIDENT = T1
           END FIND
SORT.RECS: SORT RECORDS IN FIND.RECS BY FULLNAME -
              AND EACH INCIDENT DATE
           FOR EACH RECORD IN SORT.RECS
              PRINT FULLNAME WITH INCIDENT DATE -
                AT COLUMN 25
           END FOR
END
```

produces printed output of:

```
ABBOTT, FRANKLIN G     860323
ABBOTT, GAIL H         861022
ABRAMS, RUTH Z         861115
ABRAMS, RUSSELL Y      870218
ABBOTT, FRANKLIN G     870424
ABBOTT, GAIL H         871123
      .                   .
      .                   .
      .                   .
```

If no occurrences are present

A record that does not contain at least one occurrence of the INCIDENT DATE field produces no printed output. Similarly:

```
SORT RECORDS IN FIND.RECS BY EACH A AND EACH B
```

would produce nothing for those records that do not have at least one occurrence of both A and B.

FOR EACH RECORD statement

The FOR EACH RECORD IN ORDER BY statement retrieves and loops on only the first occurrence of a field in a record.

EACH modifier

If there is more than one value of a field in a record, the special modifier, EACH, can be used to retrieve and loop on all values of the field.

The EACH modifier only can be used on a FOR EACH RECORD statement that specifies index order processing (the IN ORDER BY fieldname clause must be used and the field must have the ORDERED attribute). The VALUE IN phrase must be used to retrieve the current value of the ORDERED field driving the loop.

Example This example of the FOR EACH RECORD IN ORDER BY statement:

```
BEGIN
FR1:  FOR EACH RECORD IN ORDER BY EACH INCIDENT DATE
         PRINT VALUE IN FR1 WITH FULLNAME AT COLUMN 25
      END FOR
END
```

returns each record in order by each value of the ORDERED field, as follows:

```
760323          ABBOTT, FRANKLIN G
761022          ABBOTT, GAIL H
761115          ABRAMS, RUTH Z
770218          ABRAMS, RUSSELL Y
   .
   .
   .
```

Special statements for multiply occurring fields

User Language provides several statements for handling multiply occurring fields.

- COUNT OCCURRENCES (used with singly occurring fields, too)
- FOR EACH OCCURRENCE (used with singly occurring fields, too)
- DELETE EACH OCCURRENCE

Note: See page 19-24 for a discussion of the INSERT statement, which you can use to add new occurrences of a field.

COUNT OCCURRENCES OF statement

The COUNT OCCURRENCES OF (CTO) statement counts the number of occurrences of the named field in the current record.

Syntax

The format of the COUNT OCCURRENCES statement is:

```
label: COUNT OCCURRENCES OF fieldname
```

or

```
label: CTO fieldname
```

The field name cannot be subscripted. The COUNT OCCURRENCES OF statement can appear only within a FOR EACH RECORD loop.

The COUNT OCCURRENCES statement is supported in remote file and scattered group contexts.

COUNT IN clause

The COUNT IN clause refers to the count obtained by the COUNT OCCURRENCES OF statement. This request:

```
BEGIN
FIND.RECS:  IN VEHICLES FIND ALL RECORDS
            END FIND
            FOR 5 RECORDS IN FIND.RECS
NO.OF.VINS:    COUNT OCCURRENCES OF VIN
               PRINT OWNER POLICY WITH ’ INSURES ’ WITH -
                  COUNT IN NO.OF.VINS WITH ’ VEHICLE(S)’
            END FOR
END
```

generates the following output:

```
100025 INSURES 1 VEHICLE(S)
100030 INSURES 1 VEHICLE(S)
100032 INSURES 1 VEHICLE(S)
```

```
100051 INSURES 1 VEHICLE(S)
100058 INSURES 1 VEHICLE(S)
```

FOR EACH OCCURRENCE OF loops

On each loop of FOR EACH OCCURRENCE OF (FEO), the VALUE IN and OCCURRENCE IN labels refer to value and position, respectively, of the next field occurrence, starting with occurrence 1 and increasing by 1 for each pass through the loop. When the next field occurrence number is greater than the number of field occurrences in the record, the loop terminates.

Syntax

The format of the FOR EACH OCCURRENCE loop is:

```
label: FOR EACH OCCURRENCE OF fieldname
```

Alternatively,

```
label: FEO fieldname
```

The FOR EACH OCCURRENCE OF statement is supported in remote file and scattered group contexts.

Example

Consider this record:

```
FIRST NAME = RICHARD
LAST NAME = SMITH
CHILD = HENRY
CHILD = SALLY
CHILD = JANE
ADDRESS = AVON DRIVE
```

This request creates a separate record for each child.

```
BEGIN
FD.REC:      FIND ALL RECORDS FOR WHICH
                FIRST NAME = RICHARD
                LAST NAME = SMITH
             END FIND
             FOR EACH RECORD IN FD.REC
NOTE.ADD:       NOTE ADDRESS
CHILD.LOOP:     FOR EACH OCCURRENCE OF CHILD
                   PRINT VALUE IN CHILD.LOOP
                   STORE RECORD
                     FIRST NAME = VALUE IN CHILD.LOOP
                      LAST NAME = SMITH
                      ADDRESS = VALUE IN NOTE.ADD
                   END STORE
                END FOR
             END FOR
END
```

Simulating a FOR EACH VALUE loop

The FOR EACH OCCURRENCE OF statement can be used to simulate a FOR EACH VALUE loop if the field contains a static collection of known values. Consider the values of states. The user sets up a single record that has a multiply occurring field in sorted sequence as follows:

```
TYPE = STATE
CODE = ALABAMA
CODE = ARKANSAS
     .
     .
CODE = WYOMING
```

These statements would begin a request to generate a report in order by state:

```
BEGIN
FIND.TYPE: FIND ALL RECORDS FOR WHICH
              TYPE = STATE
           END FIND
           FOR EACH RECORD IN FIND.TYPE
EACH.CODE:    FOR EACH OCCURRENCE OF CODE
FIND.STATE:      FIND ALL RECORDS FOR WHICH
                    STATE = VALUE IN EACH.CODE
                 END FIND
                 FOR EACH RECORD IN FIND.STATE
                     .
                     .
```

This method has the advantage of eliminating the internal sort required by FOR EACH VALUE IN ORDER and provides an easy way to simulate a FOR EACH VALUE loop for a field that does not have the FRV or ORDERED attribute. However, a change in the list of values can require a recreation of the TYPE record to keep the values in order. Recreation is not required if the UPDATE IN PLACE field attribute has been specified for the CODE field and if the new and old values occupy the same place in order. Recreation also is not required if the INSERT statement (see page 19-24) is used correctly.

Deleting occurrences

A FOR EACH OCCURRENCE OF loop can be used to delete every other occurrence of a multiply occurring field as explained below. The DELETE EACH statement can be used to delete all occurrences.

Processing

The FOR EACH OCCURRENCE OF loop, when used to delete occurrences, deletes every other occurrence because the fields and records in Table B are shifted to eliminate the space the deleted occurrence took (condensing Table B storage) and the pointer is also shifted, so that the pointer ends up pointing to the third occurrence.

Example

Suppose the following statement is specified for the RICHARD SMITH record described previously:

```
DEL.CHILD: FOR EACH RECORD IN FIND.RECS
              FOR EACH OCCURRENCE OF CHILD
                 DELETE CHILD
              END FOR
           END FOR
```

The record originally contained these CHILD entries:

```
CHILD=HENRY
CHILD=SALLY
CHILD=JANE
```

On the first pass through the loop, the first value, HENRY, is selected and deleted. At the end of the first pass, the record contains these CHILD entries:

```
CHILD=SALLY
CHILD=JANE
```

On the second pass through the loop, the DELETE CHILD statement again deletes the first occurrence of the field as described on page 19-10. Since HENRY has already been deleted, Model 204 deletes the first entry, SALLY.

After the second pass, the FOR EACH OCCURRENCE loop terminates. This is because the value in DEL.CHILD should be the next (third) occurrence, but after two passes, only one occurrence remains on the record. Therefore, at the end of the FOR EACH OCCURRENCE loop, the remaining value in the CHILD field is:

```
CHILD=JANE
```

VALUE IN with FOR EACH OCCURRENCE loops

VALUE IN references to FOR EACH OCCURRENCE (FEO) statements from outside the FEO loop does not compile. Such references receive the following error message:

```
M204.0311 UNACCEPTABLE STATEMENT REFERENCE
```

At runtime, the space occupied in STBL by the FEO value is reclaimed after each iteration of the FEO loop (including the last). This results in a reduction in the runtime STBL space requirements of some programs that use FEO.

To avoid getting that error, move the value into a %variable inside the FEO loop, and then reference the %variable outside the FEO loop.

FOR EACH OCCURRENCE OF against INVISIBLE fields

Using FOR EACH OCCURRENCE (FEO) syntax against INVISIBLE fields is a waste of processing time. An FEO causes a scan of the current Table B record, but INVISIBLE fields are not stored in Table B, so an FEO against an INVISIBLE field can never find any occurrences. Prior to V4R2.0 this programming flaw was masked because using FEO syntax against an

INVISIBLE field compiled and evaluated successfully, although it never found occurrences.

Beginning in V4R2.0, FEO syntax against an INVISIBLE field results in compilation error:

```
M204.0320 FIELD IS INVISIBLE. FIELD =
```

UPDATE field attribute

Impact of changing a value

When the user changes the value of a field, how Model 204 changes the occurrence depends upon whether the field was defined with the UPDATE IN PLACE or UPDATE AT END attribute, as follows:

Attribute	How the update works…
UPDATE IN PLACE (the default)	The value of the field occurrence is changed but its position in the record is preserved. To change the order of values, the user must delete the old value and add the new one in separate statements.
UPDATE AT END	The existing occurrence is deleted and the new one is automatically added at the end. UPDATE AT END is normally specified for applications that depend on value rotation to accomplish aging.

Example

This example includes both approaches to updating. Suppose a record has the following fields:

```
NAME = RICHARD SMITH
CHILD = HENRY
CHILD = SALLY
CHILD = JANE
```

You could use the technique illustrated below to add a last name to each child. (This technique involves the use of %variables, which are discussed in Chapter 10.)

```
BEGIN
FIND.RECS:  FIND ALL RECORDS FOR WHICH
               NAME = RICHARD SMITH
            END FIND
            FOR EACH RECORD IN FIND.RECS
EACH.CHILD     FOR EACH OCCURRENCE OF CHILD
                  %A = VALUE IN EACH.CHILD WITH ’ SMITH’
                  CHANGE CHILD = VALUE IN EACH.CHILD TO %A
               END FOR
            END FOR
END
```

If the UPDATE IN PLACE option is specified

If the UPDATE IN PLACE option has been specified for the CHILD field, then the FOR EACH OCCURRENCE loop change each occurrence of CHILD in turn.

On the first pass through the loop, the first value, HENRY, is selected and changed to HENRY SMITH. After the first pass, the record looks like this:

```
NAME = RICHARD SMITH
CHILD = HENRY SMITH
CHILD = SALLY
CHILD = JANE
```

At the end of the second pass, SALLY is changed:

```
NAME = RICHARD SMITH
CHILD = HENRY SMITH
CHILD = SALLY SMITH
CHILD = JANE
```

At the end of the third pass, JANE is changed:

```
NAME = RICHARD SMITH
CHILD = HENRY SMITH
CHILD = SALLY SMITH
CHILD = JANE SMITH
```

If the UPDATE AT END option is specified

If the UPDATE AT END option has been specified for the CHILD field, the FOR EACH OCCURRENCE loop proceeds as described here.

On the first pass through the FOR EACH OCCURRENCE loop the first value, HENRY, is selected and changed to HENRY SMITH. The act of changing, however, causes the value HENRY to be deleted and the value HENRY SMITH to be added as the last child. Thus after the first pass, the record looks like this:

```
NAME = RICHARD SMITH
CHILD = SALLY
CHILD = JANE
CHILD = HENRY SMITH
```

On the second pass through the loop, JANE, which is now the second occurrence of CHILD, is deleted and JANE SMITH added to the end of the record:

```
NAME = RICHARD SMITH
CHILD = SALLY
CHILD = HENRY SMITH
CHILD = JANE SMITH
```

On the third pass, the third occurrence is JANE SMITH, and the record ends with:

```
NAME = RICHARD SMITH
CHILD = SALLY
CHILD = HENRY SMITH
CHILD = JANE SMITH
```

Subscripts

Subscripts can be included in Model 204 field references to facilitate the selection of particular occurrences of multiply occurring fields. Any field name can be followed by a parenthesized expression. The value of this expression is used as an ordinal number which specifies the desired occurrence of the named field. For example:

```
INCIDENT(3)
COURSE.NUMBER(2)
TRANSACTION(A+B)
```

Example This request illustrates a class schedule where subscripts are used to change the room number for a course:

```
BEGIN
COURSE:  FIND ALL RECORDS FOR WHICH
            REC = ROOM ASSIGNMENT
         END FIND
CHANGE:  FOR EACH RECORD IN COURSE
            IF ROOM(1) EQ ’214A’ THEN
               CHANGE ROOM(1) TO ’566A’
            END IF
            IF ROOM(2) EQ ’214A’ THEN
               CHANGE ROOM(2) TO ’566A’
            END IF
         END FOR
END
```

Subscripted field extraction

Subscripted field references attempt to maintain their position inside a record much as an FEO loop attempts to maintain its position inside a record. This means that subscripted field references tend to be as efficient as FEO loops.

The following is an example of using subscripted field extraction for a repeating field group:

```
%INC IS STRING ARRAY (12) NO FS
%IDATE IS STRING ARRAY (12) NO FS
FEOIDATA: FEO INCIDENT
  %INC(OCCURRENCE IN FEOIDATA) = VALUE IN FEOIDATA
%IDATE(OCCURRENCE IN FEOIDATA) = INCIDENT
DATE(OCCURRENCE IN FEOIDATA)
  END FOR
```

If you use multiple FEO loops for field group processing, it is possible that using this technique will require additional VTBL resources for procedures with an extremely large number of sorts or subscripted field references.

Evaluation of subscript expressions

The evaluation of subscript expressions is subject to the rules for determining an integer result for an arithmetic expression as described in “Arithmetic operations” on page 10-3.

Statements and phrases with which you cannot use subscripts

Subscripted field references can appear anywhere that unsubscripted references can, *except* in the following statements and phrases:

Statements you cannot use with subscripts	**Because…**
ADD	Adds a new occurrence of a field.
BY EACH phrase in the SORT RECORDS	Loops through all occurrences of a field.
DELETE EACH statement	Loops through all occurrences of a field.
EACH phrase in a PRINT specification	Loops through all occurrences of a field.
FILE	Deals with fields having the INVISIBLE attribute, which cannot be the object of subscripted references.
FIND	Locates records without regard to which occurrence of a field contains the desired value.
FOR EACH OCCURRENCE	Loops through all occurrences of a field. Field references within FOR EACH OCCURRENCE loops can be subscripted, depending upon the individual statements in which the references appear.
FOR EACH VALUE	Loops through all values of all occurrences of the specified field. Field references within FOR EACH VALUE loops are not allowed at all, unless the field reference is embedded in a nested FOR EACH RECORD loop. See “Setting up a value loop on one field and printing a value of another” on page 8-20 for more information.
IN ORDER BY EACH *phrase in the* FOR EACH RECORD	Loops through all occurrences of a field.
PRINT *n*	Loops through all occurrences of a field.
STORE RECORD	Adds fields in the order of appearance in the statement.

Unsubscripted field references

An unsubscripted field reference in a context in which subscripted references are allowed is always equivalent to a subscripted reference with a value of one.

Do not use subscripts with INVISIBLE fields

You cannot make subscripted references to fields that have the INVISIBLE attribute (see the discussion on field attributes in Chapter 32). These fields are not truly multiply occurring, although they can have several different values in a single record. A subscript specified for a field with the INVISIBLE attribute is ignored.

Subscript validity rules

The rules presented below indicate whether or not a subscript value is valid and what action to take if the value is not valid. These rules take into account:

- The value of the subscript
- The context in which the subscript appears
- The description of the subscripted field

Explanation of the rules

In these rules, two quantities are used:

1. *N* is the maximum number of occurrences that can be stored in a record for a given field. For a preallocated field, N equals the value of n in the OCCURS clause of the field's description. For other fields, N has no limit.
2. *P* is the number of nonempty occurrences of the referenced field found in the specified record when the reference is evaluated.

For a summary of rules for preallocated fields, refer to the discussion on preallocated fields in "Storing values in preallocated fields" on page 15-16.

INSERT statement

The INSERT statement, like the ADD statement, is used for adding new occurrences of a field. INSERT is used to add occurrences when the order of the values is important and the values are added out of order.

The INSERT statement is supported in remote file and scattered group contexts.

Syntax

The format of the INSERT statement is:

```
INSERT fieldname [ (subscript)] = value
```

Example

Assume a record with these fields:

```
DEPT = PERSONNEL
DEPT = FINANCE
DEPT = MARKETING
```

The following statement:

```
INSERT DEPT(3) = ACCOUNTING
```

results in the record:

```
DEPT = PERSONNEL
DEPT = FINANCE
DEPT = ACCOUNTING
DEPT = MARKETING
```

Subscript validity rules

Table 19-1 lists validity rules for subscripts. In this table:

- P is the number of occurrences of the field in the record when the INSERT statement is issued.
- S is the subscript specified in the INSERT statement.

Table 19-1. Subscript validation rules

If P and S values are:	Then Model 204 takes this action:
P > S	Inserts the new occurrence in front of the former Sth occurrence; the new occurrence becomes the current Sth occurrence of the field.
P < S	Inserts the new occurrence of the field after the Pth occurrence; the new occurrence becomes the (P+1) occurrence.
P = 0	Adds the new value at the end of the record, as in the ADD statement.
S = 0, or no subscript	Treats the new value as if S = 1 and inserts the new value as the first occurrence, in front of any former occurrence of the field.
S < 0	Does not add a new occurrence.

For fields with the INVISIBLE attribute, only the index is affected.

PRINT statement

In retrieval statements such as PRINT, subscript values less than zero or greater than P are invalid. Invalid references of this kind cause the null value to be returned. A subscript of zero returns the value of the first occurrence.

DELETE statement

In the DELETE statement, subscript values less than one or greater than P are invalid. If an invalid reference of this kind is made, no action is taken.

If several occurrences of a field are being deleted, you should be careful *not* to use DELETE in the following way:

```
FOR EACH RECORD IN FIND.RECS
   DELETE CLIENT(1)
   DELETE CLIENT(2)
   DELETE CLIENT(3)
```

```
END FOR
```

As the statements are executed, Model 204 deletes the first occurrence, CLIENT(1), then locates the current second occurrence, which is the original CLIENT(3), and deletes it. Then, because a third occurrence cannot be found, the operation stops, and the original CLIENT(2) is never deleted.

In such a situation, deleting the occurrences in the reverse order achieves the desired result:

```
FOR EACH RECORD IN FIND.RECS
   DELETE CLIENT(3)
   DELETE CLIENT(2)
   DELETE CLIENT(1)
END FOR
```

The desired result also can be achieved by completely omitting the subscripts, as follows:

```
FOR EACH RECORD IN FIND.RECS
   DELETE CLIENT
   DELETE CLIENT
   DELETE CLIENT
END FOR
```

CHANGE statement

In the CHANGE statement, subscript values less than one or greater than P are treated as attempts to add a new occurrence (P+1). If (P+1) does not exceed N (the maximum number of occurrences that can be stored), the new occurrence is added to the record. If (P+1) does exceed N, the request is cancelled.

SORT RECORDS statement

The use of subscripts in references to the record set yielded by a SORT RECORDS statement sometimes can produce unexpected results. If the BY EACH option appears in a SORT statement, records in which the BY EACH field is multiply occurring are copied several times (once for each field occurrence) to the system scratch file CCATEMP. In each copy, the occurrences of the BY EACH field are rotated, so that the occurrence used as the sort key appears first. Therefore, a subscripted reference to this field can yield different values for different copies of the record.

20

Global Features

In this chapter

Overview

Model 204 offers several global features to store information in memory so that it is not automatically cleared between requests. The memory area for storing global information is a Model 204 internal work area or server table called global table or GTBL.

The global features are:

- Global string variables
- Global objects
  - Global found sets and lists
  - Global screens, images, and menus
  - Global positions

Global information is available only to the user who creates it.

GTBL internal work area

Each user’s GTBL is empty when the user logs in. GTBL accumulates global information that is available for the duration of the terminal session, unless you intentionally clear it. You can clear GTBL information selectively.

For a discussion of GTBL space requirements, see “GTBL (global variable table)” on page 21-6.

Global items are stored in a specific order in GTBL. As shown in Figure 20-1 on page 20-3, the area that stores global string variables is at the beginning of the table, and is built from the top down. The area that stores global objects is at the end of the table and is built from the bottom up. The unused or free space is between these two areas.

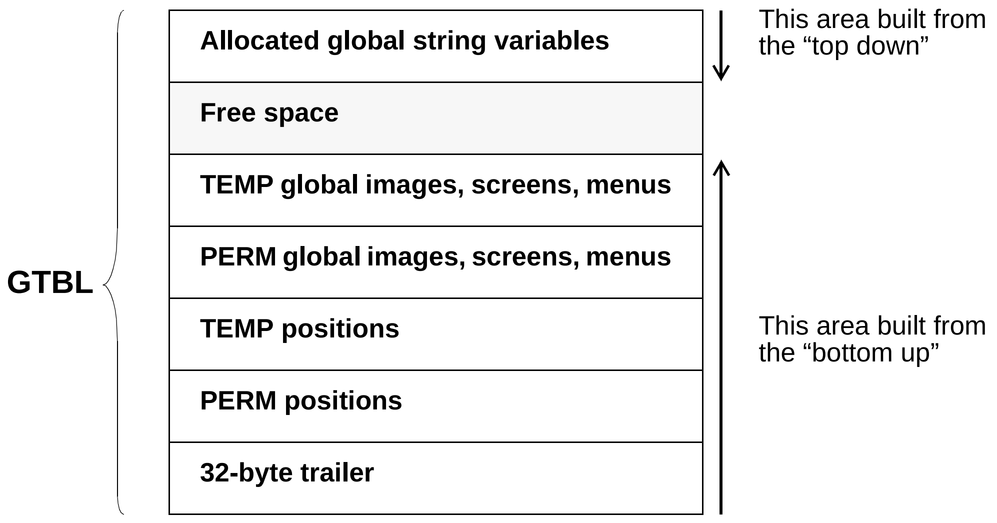

GTBL
Allocated global string variables
Free space
TEMP global images, screens, menus
PERM global images, screens, menus
TEMP positions
PERM positions
32-byte trailer
This area built from the “top down”
This area built from the “bottom up”

Figure 20-1. Storage of global variables and global objects in GTBL

Global string variables

You can use global string variables to:

- Pass information from one request to another request
- Include procedures conditionally at the Model 204 command level
- Tailor a request dynamically

As of Version 5.1, you can increase the speed and reduce the CPU time to find and update a global string variable by setting the GTBLHASH parameter to a nonzero value. The GTBLHASH parameter specifies the number of buckets allocated in the global string variable section of GTBL. When GTBLHASH is a nonzero value, and you set or get a global string variable, the global string variable name is hashed to determine the bucket in which the name is located.

This reduces the overall amount of data that must be scanned to find a global string variable or must be moved when a value is deleted or changes in size. If GTBLHASH=0, global string variables are processed as in pre-5.1 versions of Model 204. See "GTBLHASH: Number of bucket for global string variables" in the *Model 204 Command Reference Manual.*

The GTBLPCT parameter determines the initial percentage of GTBL to allocate for global string variables. The default value of GTBLPCT is 50, meaning 50 percent of GTBL is initially allocated for global variable strings. The remainder, in this case 50 percent, is the initial allocation for global objects. However, if GTBLHASH=0, a nonzero setting for GTBLPCT has no effect.

When in effect, if either area of GTBL fills and there are still free pages in GTBL, then GTBL can be rearranged if more space is required in the full area of GTBL.

Because these rearrangements can be CPU intensive, CCA recommends that you determine an accurate setting for GTBLPCT to avoid frequent rearrangements. You can monitor the performance of the hash GTBL feature using the GTBLRU user statistic and the GTBLRS since-last statistic. See "GTBLPCT: Initial percentage of GTBL to allocate for global string variables" in the *Model 204 Command Reference Manual.*

Rearranging GTBL and tracking the rearrangements

The following statistics are available as system statistics, user statistics and since-last statistics to keep track of GTBL rearrangements required for the hashed GTBL feature:

Statistic	Tracking
GTBLRU	Number of GTBL rearrangements required to add a string variable global.
GTBLRS	Number of GTBL rearrangements required to add a global object.

After reviewing the GTBLRU and GTBLRS statistics, you can consider taking the following actions:

- If both of these values are high, increase the size of GTBL by increasing LGTBL.
- If GTBLRU is high but GTBLRS is not, increase GTBLPCT or decrease GTBLHASH.
- If GTBLRS is high but GTBLRU is low, decrease GTBLPCT.

Global found sets and lists

You can use global found sets and lists to make found sets and lists available across request boundaries. A global found set or list remains in GTBL until you:

- Explicitly delete it
- Issue a RELEASE or COMMIT RELEASE statement
- The file or group it refers to is closed, including the file close processing done when exiting a subsystem or stopping
- Log out

Sorted sets are a subgroup of found sets; they are treated the same by the CLEAR statement that is used to clear global found sets.

Global positions

You can use REMEMBER and POSITION statements to save and recall a place in a FOR loop, either globally or nonglobally. This lets you suspend processing in a FOR loop and resume it later, either within the same request or in a subsequent request.

Global images and screens

You can use global images and screens to:

- Pass image and screen data from one request to another
- Manage more than one image or screen in one request
- Manage menus, which are a special type of screen
- Reduce I/O by swapping modified global screens to CCASERVR instead of paging them between the buffer pool and CCATEMP

Global string variables

Global string variable names and values

Entries in the GTBL consist of global name=value pairs. The names and values of the global string variables are created with the $SETG function from within a request.

- A global name can consist of up to 255 characters and follows the naming conventions for variables (see “%Variable names” on page 10-7). Global string variables that contain an underscore (_) cannot be used as ?& dummy strings; use a period (.) instead.
- A value can be 0, null string, to 255 characters.

Clearing global string variables

When a new value for an existing name is stored, the old entry is first deleted. Entries remain in the table until you delete them by issuing a CLEARG command, execute a CLEAR statement, execute a $DELG function call, or you log out.

If you define a large number of global string variables, Model 204 performance can be adversely affected.

For details and examples of clearing global string variables, see “Clearing the GTBL work area” on page 20-28.

Global variable functions and commands

The following functions, statements, commands, and facilities manipulate entries in the global variable table:

- Use the functions $SETG and $GETG within a request to store and retrieve global string variables. Use $INCRG to perform simple arithmetic on global string variables with numeric values. Use the $DELG function to delete global string variables created by either $SETG or $INCRG. The format of these functions is described in Chapter 27.
- Use the ?& dummy string within a request to read a variable in the GTBL. Refer to Chapter 13 for more information about dummy strings.
- Use the conditional INCLUDE command (IF *A* = *B*,*name*) to search the global variable table for an entry whose name is *A* and whose value is *B*. If the entry is found, the named procedure is included. The IF command is discussed on “Keep IF commands at as high a nesting level as possible” on page 20-12.

Using global string variables in application subsystems

In addition to the above facilities, an application subsystem can designate specific global string variables in the subsystem definition. For more information about subsystem global string variables, refer to:

- “Command line global variable” on page 23-5
- “Communication global variable” on page 23-6
- “Error global variable” on page 23-9

Passing string values from one request to another

It is often necessary to generate data in one request and to save the data for use in other independent requests to be run later in the terminal session.

Example 1 Suppose that you want to store the current date in every record created during a session in YYMMDD format, a modified form of the value returned by the $DATE function. You can derive the date once at the beginning of the day and hold it for use throughout the day.

At the beginning of the terminal session, you enter:

```
BEGIN
    %DATE = $DATE
    %DATE = $SUBSTR(%DATE, 1, 2) -
         WITH $SUBSTR(%DATE, 4, 2) WITH -
         $SUBSTR(%DATE, 7, 2)
     IF $SETG('DATE', %DATE) THEN
          PRINT '*** REQUEST TOO LONG - GTBL'
          STOP
     END IF
END
```

Later in the terminal session, you can use the specially formatted date:

```
BEGIN
    %DATE = $GETG('DATE')
    STORE RECORD
        FIELD = VALUE
        FIELDB = VALUE
        DATE = %DATE
            .
            .
            .
    END STORE
```

Example 2 Assume that different sets of records are processed and the results are used to produce a final report. Because of compiler table limitations, you have to process requests that cannot be continued with the MORE command (see Chapter 21). You can store intermediate results in GTBL and produce the final report exclusively from the table.

For example:

```
BEGIN
          .
          .
          .
      process hourly workers' wages
          .
          .
          .
      IF $SETG('HOURLY TOTAL', %TOTAL) THEN
```

```
            PRINT '*** REQUEST TOO LONG - GTBL'
            STOP
        END IF

END

BEGIN
            .
            .
            .
        process monthly workers' wages
            .
            .
            .
        IF $SETG('MONTHLY TOTAL', %TOTAL) THEN
            PRINT '*** REQUEST TOO LONG - GTBL'
            STOP
        END IF
END

BEGIN
        %HOURLY = $GETG('HOURLY TOTAL')
        %MONTHLY = $GETG('MONTHLY TOTAL')
             .
             .
             .
        format report
             .
             .
             .
END
```

Using global string variables with a conditional INCLUDE command

You can use global string variables to create a modular programming environment in which you select procedures to perform a particular function without compiling and evaluating procedures designed for other related functions.

To use global string variables effectively in creating such an environment, it is useful to review the differences between commands and User Language statements and between conditional and unconditional includes, as described in the following sections.

Differences between commands and User Language statements

Commands and User Language statements have different effects and are used in different ways:

- System control commands can be issued only outside a request—at command level. They are acted upon immediately. User Language statements can be used only within a request. They are compiled on a line-by-line basis, but the entire request is not executed until Model 204 receives an END statement.
- INCLUDE is both a command and a User Language statement. In either context, Model 204 is directed to take the next input line from an appropriate stored procedure. When the procedure lines are exhausted, the next input line is taken from the command or User Language statement immediately following the INCLUDE.
- IF has two formats:
  - Conditional INCLUDE command
  - User Language statement

Conditional and unconditional INCLUDEs

A conditional include can be coded within an IF statement using the INCLUDE statement or with an INCLUDE command.

An unconditional include can be coded with a standalone INCLUDE statement or command.

This section describes conditional INCLUDE commands by providing examples and discussing how each example is processed.

Example 1 The following request illustrates the conditional INCLUDE command:

```
BEGIN
ALL: FIND ALL RECORDS
     END FIND
     FOR EACH RECORD IN ALL
         IF AGE GT '10' THEN
             IF $SETG('AGE', 'YES') THEN
                 PRINT '*** REQUEST TOO LONG - GTBL'
                 STOP
             END IF
         END IF
     END FOR
END
IF AGE = YES, COUNT
```

How Example 1 is processed

The statements between the BEGIN and END are compiled and evaluated. If the AGE condition is true, a global string variable is set. Model 204 then processes the IF command. Statements in the COUNT procedure are compiled and executed only if the condition is true. Otherwise, the statements are never compiled.

Example 2 Suppose that you enter record selection criteria and then select one of three reports to be generated. A set of four procedures can be created.

For example, procedure A might contain the prompts for record selection and report type. Procedures B, C, and D might contain statements to produce the individual reports. Only procedure A and one of the other three procedures is compiled and evaluated.

```
PROCEDURE A
BEGIN
FIND.RECS: FIND ALL RECORDS FOR WHICH
               ??SELECT.RECORDS
           END FIND
           IF $SETG('REPORTNUM', $READ('ENTER REPORT NO'))
               THEN PRINT '*** REQUEST TOO LONG - GTBL'
               STOP
           END IF
END MORE
IF REPORTNUM = 1, B
IF REPORTNUM = 2, C
IF REPORTNUM = 3, D
END PROCEDURE
```

Procedures B, C, and D have the same basic format, but variations in processing are applied to each record.

```
PROCEDURE B
MORE
PROCESS: FOR EACH RECORD IN FIND.RECS
             .
             .
             .
         processing
             .
```

```
                 .
                 .
            END FOR PROCESS
END
END PROCEDURE
```

How Example 2 is processed

The dialog produced by these procedures is shown below, with user input in boldface:

```
INCLUDE A

??SELECT.RECORDS

REGION = SOUTH OR WEST

$$ENTER REPORT NO

2

output from Procedure C
```

Keep IF commands at as high a nesting level as possible

An alternate method of writing procedures is to follow each report's END MORE statement with IF commands. However, each procedure might then INCLUDE itself or another procedure, creating a lower level of nesting. If the procedure continued to INCLUDE itself, the maximum nesting level would be reached. As a general rule in a complex set of procedures, keep IF commands at as high a nesting level as possible. The global string variables to be tested can be set at any level.

Using global string variables to tailor a request

Global string variables can be used in conjunction with the FILE$ condition (see "FILE$ condition" on page 4-31) to access a set of files in a group.

Example

In the following example, the value of the global string variable, FILES, can be changed to access an alternative set of files. This example consists of two requests. The first—BEGIN through END MORE—sets the global string variable during the execution phase; the second—MORE through END—is then compiled with the correct value.

```
BEGIN
               .
               .
               .
           IF $SETG('FILES','FILEA OR FILE$ FILEC')
THEN
               PRINT '***REQUEST TOO LONG -- GTBL'
               %IGNORE = $SETG('FILES', '')
           END IF
END MORE
MORE
           IF $GETG('files') = '' THEN
               STOP
           END IF
 GET.A:    FIND ALL RECORDS FOR WHICH
              (FILE$ ?&FILES) AND FIELDX = 'A'
           END FIND
           FOR EACH RECORD IN GET.A
               PRINT ALL INFORMATION
           END FOR
END
```

Global objects

Global objects include found sets, images, lists, menus, positions, screens, and sorted sets.

General rules for declarations

- You must declare global lists, found sets, and sorted sets in every request that references them; the DECLARE statement must come before the reference.
- If a global object is used in multiple subroutines, or in both the main program and a subroutine, the label must be declared as global in the main program before the subroutine(s). The subroutine(s) must then declare the label as common.
- If you refer to a global object before a DECLARE statement, the system issues a duplicate label compilation error. Such a reference implicitly makes the object nonglobal.

Incompatibility

The PQO product does not support global found sets, lists, positions, or sorted sets. These global objects cannot be used in conjunction with remote files or scattered groups. This limitation exists, because GTBL does not exist on the PQO server.

Clearing global objects from GTBL

For syntax, details and examples of clearing global objects, see “Clearing the GTBL work area” on page 20-28.

Using global found sets and lists

You can pass found sets and lists from request to request by declaring them as GLOBAL. A global found set or list is stored in the internal work area GTBL for the duration of the terminal session unless it is intentionally cleared, or the file or group with which it is associated is closed.

Creating a global found set or sorted set

You can create a global found set using this syntax:

Syntax

```
[DECLARE] LABEL labelname [GLOBAL | COMMON]
```

Where *labelname* is a unique global object name.

Creating a global list

You can declare a global list using the following syntax:

Syntax

```
[DECLARE] LIST listname

  [IN [FILE [PERM | TEMP] GROUP] name]

  [GLOBAL | COMMON]
```

In both cases, the keyword GLOBAL implies COMMON. These two keywords are mutually exclusive in the declaration statement.

- Because GLOBAL implies COMMON, a list or found set can be declared GLOBAL at any scope. See “Scope of elements” on page 12-15 for a discussion.
- The global name, *labelname*, or *listname*, must be unique across all global objects. For a discussion, see “Sharing common elements” on page 12-15.

Usage rules

The following rules apply to global found sets and lists:

- You must declare each global list or global found set as *labelname* GLOBAL in each procedure that uses it.
- The file context in which a global list or found set is used must be the same as the file context in which it was created.

  The Model 204 compiler does not currently enforce this rule; therefore, your code must maintain this requirement. CCA recommends that you populate a global found set or list only once within the scope of these global objects. However, this is not a restriction as long as file context is maintained.
- Global found sets and lists are invalid in ad hoc group context.
- Each global object name must be unique.

- Global found sets and lists are *not* supported in remote file or scattered group contexts.

Example 1: Referencing a global found set

In this example, Procedure 1 declares the global found set and then performs the find. Procedure 2 needs only to declare the global found set and then reference the global found set labeled F1. The FOR loops in both procedures process records in the VEHICLES file for which the value of the field COLOR was equal to BLUE at the time of the evaluation of the FIND command in Procedure 1.

Procedure 1

```
BEGIN
DECLARE LABEL F1 GLOBAL
F1: IN VEHICLES FIND ALL RECORDS WHERE COLOR = 'BLUE'
    END FIND
    FOR EACH RECORD IN F1
       PAI
    END FOR
END
```

Procedure 2

```
BEGIN
DECLARE LABEL F1 GLOBAL
    FOR EACH RECORD IN F1
       PAI
    END FOR
END
```

Example 2: Maintaining file context

Example 2 illustrates repopulating a global found set across the scope of its usage, maintaining the same file context throughout. In this example, Procedure 1 is identical to Procedure 1 in Example 1 above, and Procedure 3 is identical to Procedure 2 in Example 1. The second procedure in this example, however, performs a different find than the first. Procedure 3 then processes the records found in Procedure 2.

So in this example, Procedure 1 processes records in the VEHICLES file for which the value of the field COLOR is equal to BLUE, while Procedures 2 and 3 process records for which COLOR was equal to RED at the time of the FIND command evaluation in Procedure 2.

Procedure 1

```
BEGIN
DECLARE LABEL F1 GLOBAL
F1: IN VEHICLES FIND ALL RECORDS WHERE COLOR = 'BLUE'
    END FIND
    FOR EACH RECORD IN F1
       PAI
    END FOR
END
```

Procedure 2

```
BEGIN
DECLARE LABEL F1 GLOBAL
F1: IN VEHICLES FIND ALL RECORDS WHERE COLOR = 'RED'
    END FIND
    FOR EACH RECORD IN F1
       PAI
    END FOR
END
```

Procedure 3

```
BEGIN
DECLARE LABEL F1 GLOBAL
    FOR EACH RECORD IN F1
       PAI
    END FOR
END
```

Using global sorted sets

A global sorted set is created when a SORT RECORDS statement or a FOR EACH RECORD IN ORDER BY statement is preceded by a label that has been declared GLOBAL.

Limiting subsequent references

For example, in Procedure 1 below, MAKE is the only field referenced in the request. Therefore, MAKE is the only field that can be referenced in subsequent requests. Thus, in Procedure 2, because the field MODEL was not referenced in the previous request, a blank is printed each time through the FOR loop.

Procedure 1

```
BEGIN
DECLARE LABEL S1 GLOBAL
F1: IN VEHICLES FIND ALL RECORDS
    END FIND
S1: SORT RECORDS IN F1 BY MAKE
    FOR EACH RECORD IN S1
       PRINT MAKE
    END FOR
END
```

Procedure 2

```
BEGIN
DECLARE LABEL S1 GLOBAL
    FOR EACH RECORD IN S1
       PRINT MODEL
    END FOR
END
```

Keeping all fields accessible for subsequent references

To ensure that all fields are accessible to subsequent requests, use a PAI or field name variable with the statement that creates the found set.

The sort key field(s) used on the SORT statement cannot be referred to by a subsequent request unless referred to in the SORT statement FOR loop in the request creating the set. While this restriction applies to the use of the SORT RECORDS statement, it does *not* apply to the use of the SORT RECORD KEYS statement.

If you are not going to refer to the sorted set in the request that creates the sorted set, you can code a FOR loop, which is compiled but never executed, that refers to each field that you want to refer to in a subsequent request.

Saving and recalling a POSITION in a FOR loop

You can include statements in a FOR loop to provide for the possibility of terminating the loop before the set of records or values being processed is exhausted. For example, you might use a JUMP TO or LOOP END statement in conjunction with an IF statement to test each record or value before processing, and terminating the loop if a certain condition is met.

The REMEMBER and POSITION statements let you store the current processing position in a FOR loop, then recall it at a later time and resume FOR processing where you left off earlier.

REMEMBER statement

The purpose of the REMEMBER statement is to store the processing position in a FOR loop. Each REMEMBER statement creates a GTBL entry of variable length. See Chapter 21 for detailed descriptions of GTBL entries. If you remember a position as GLOBAL, the entry remains in GTBL after the current request ends. Model 204 clears nonglobal REMEMBER positions at the end of each request.

Syntax

```
REMEMBER [GLOBAL] position_name
   [IN foundsortset_name | ON list_name]
```

Where

- The *GLOBAL* option retains the REMEMBER position after the current request terminates. If GLOBAL is not specified, the position is temporary, and is cleared from GTBL after the current request terminates.
- *position_name* is a unique object name you assign to the REMEMBER position.
- The *foundsortset_name* or *list_name* must be identical to the listname or label specified on the FOR loop:
  - If you are processing a found set or a sorted set, IN *foundsortset_name* is the label of the FIND statement that generated it.
  - If you are processing a list, ON *list_name* is the name you gave to the list when you originally declared it.

Usage notes

The following rules apply to the use of the REMEMBER statement:

- Can appear only within a FOR loop, but not a nested FOR loop. The file context must be identical to the context of the FOR loop.
- Is invalid in ad hoc group context, if used with the GLOBAL option.
- The REMEMBER statement is incompatible with FOR EACH RECORD IN ORDER BY clauses.

POSITION statement

A POSITION statement recalls a remembered position, so you can resume FOR processing in a list or found set that was terminated at an earlier time. For example, if you remembered the position at record number 5, then your foundset is positioned at record number 6 for further processing.

This POSITION statement is not to be confused with the POSITION statement used to read records sequentially in an external VSAM KSDS file. See “POSITION statement” on page 17-42.

Syntax

```
POSITION {FOUNDSET foundsortset_name

  | LIST list_name} [AT] position_name
```

Where

- The *foundsortset_name* or *list_name* must be identical to the list or label name used on a previous REMEMBER statement and in the FOR statement that follows.
  - If you are about to resume the processing of a found set, *foundsortset_name* is the label of the FIND statement that generated it.
  - If you are about to resume the processing of a list, *list_name* is the name you gave to the list when you originally declared it.
- *position_name* is a name that you assigned to the remembered position when you stored it on a previous REMEMBER statement.

Usage notes

The following rules apply:

- The POSITION statement must appear outside, immediately before the FOR loop to which it refers; it cannot be in the FOR loop itself.
- The FOR statement that follows the POSITION statement must have an identical file context and list name or label name as the statement that initiated the FOR loop of the remembered position.
- The POSITION statement is incompatible with FOR EACH RECORD IN ORDER BY clauses.

Global images and screens

You can use global images and screens to pass image and screen data from one request to another and to efficiently manage more than one image or screen in one request.

You can also declare global menus, because a menu is a special type of screen. In this section, the term “screen” applies to both screens and menus, unless otherwise noted.

Declaring global images and screens

You make an image or screen global by specifying the keyword GLOBAL in its declaration statement. You can declare global images and screens to be TEMPORARY, deleted at request termination, or PERMANENT, persist across request termination.

How images and screens are processed

When you declare an image or screen as GLOBAL, it is stored in GTBL. Nonglobal images and screens, declared as COMMON, or neither COMMON nor GLOBAL, are stored in FSCB.

This section provides some background information about how Model 204 processes images and screens, contrasting how global and nonglobal images and screens are processed.

Images and screen processing

Figure 20-2 and Figure 20-3 represent how nonglobal and global objects are processed. Comparing the two types of processing highlights the I/O and storage savings associated with the global images and screens

Nonglobal object processing.

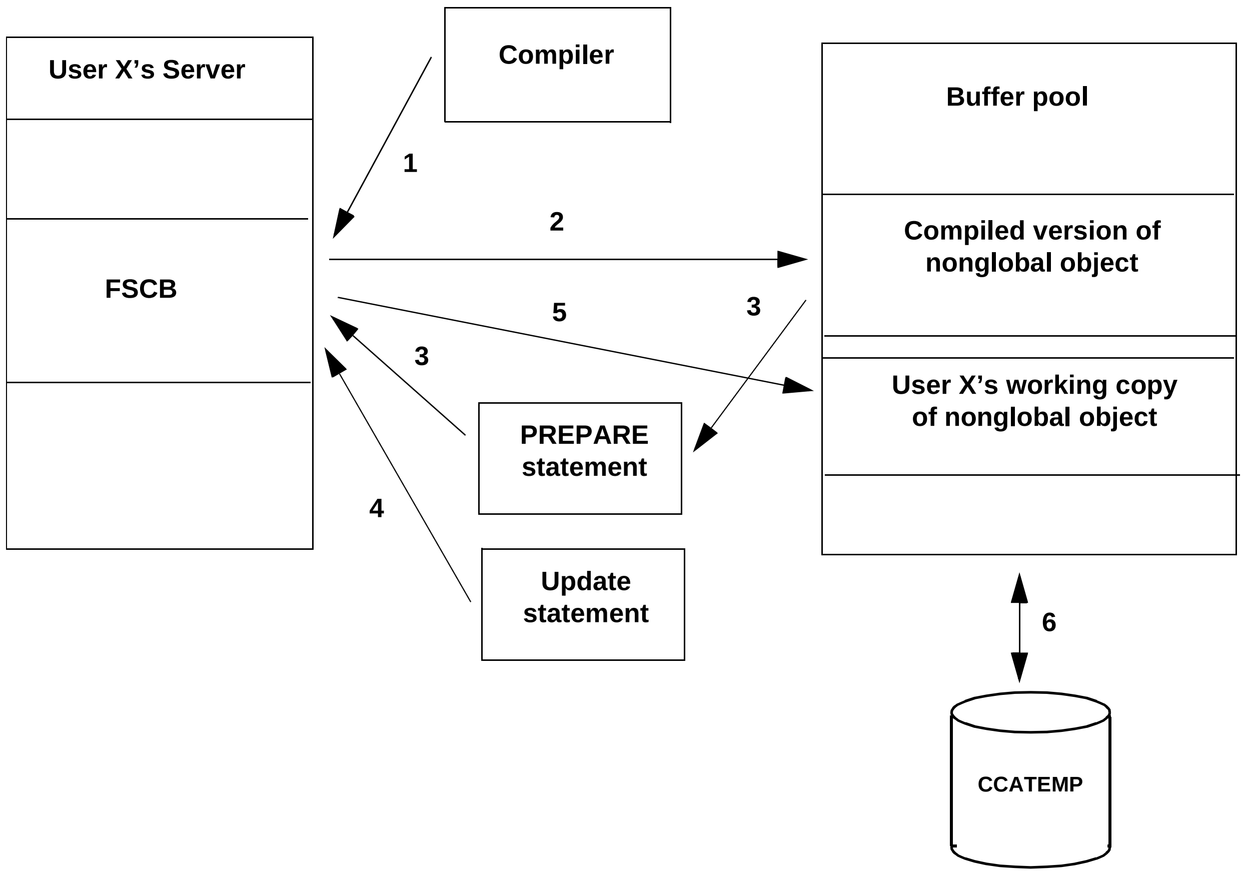


Figure 20-2. Processing of nonglobal objects

Figure 20-2 illustrates the following process in steps:

1. The compiler loads a compiled version of the screen or image into FSCB in the user's server.
2. A copy of the compiled object is stored in the buffer pool.
3. A PREPARE statement acts on the compiled object in the buffer pool and copies it into FSCB.
4. Update statements act on the copy of the object in FSCB.
5. The updated object is copied from FSCB to a working copy of the object in the buffer pool.
6. If a user waits too long or the buffer pool fills up, the object is written to CCATEMP. When a user presses ENTER for that object, the object is loaded back from CCATEMP to the buffer pool.

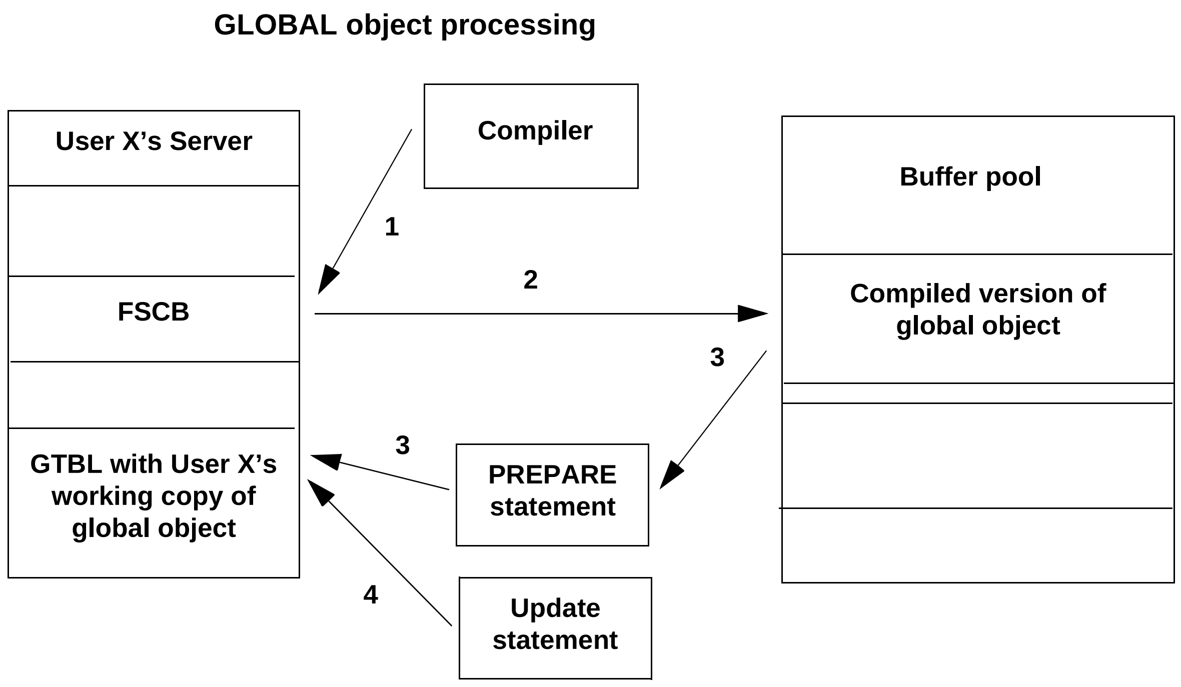


Figure 20-3. Processing of global objects

Figure 20-3 illustrates the processing in steps:

1. The compiler loads a compiled version of the screen or image into FSCB in the user's server.
2. A copy of the compiled object is stored in the buffer pool. This copy is called 'VIRGIN', as it is never updated.
3. A PREPARE statement acts on the compiled object in the buffer pool and copies it into GTBL.
4. Update statements act on the copy in GTBL.

Using PREPARE and IDENTIFY statements

To help you decide how to use the PREPARE and IDENTIFY statements review the following:

- "Storage of global variables and global objects in GTBL" on page 20-3
- "Timing and placement of a CLEAR statement" on page 20-31
- "Performance considerations" on page 20-32
- For syntax and usage of the PREPARE statement, see "PREPARE statement" on page 22-26
- For syntax and usage of the IDENTIFY statement, see "IDENTIFY statement" on page 17-38

Also when using the IDENTIFY statement, see “Consistency checks performed” on page 20-24.

Performance and efficiency benefits

Global images and screens can simply and efficiently perform image-to-image processing and screen to image processing, eliminating the need to map screen items to image items when passing or preserving screens. Using global images and screens:

- Reduces CPU resources required to pass data between requests in applications where a large number of global variables would have been necessary.
- Reduces the FSCB I/O between multiple images and between images and screens.
- Makes the declaration of global data explicit, thus traceable by using naming conventions and cross-reference tools.
- Simplifies code, eliminating the need for `IF $SETG() THEN …` sequences to map request %variables into global variables and for `%XYZ = $GETG()` sequences to map global variables into request %variables.
- Reduces I/O by swapping modified global screens to CCASERVR instead of paging them between the buffer pool and CCATEMP.

When to use global images and screens

You achieve the greatest benefit by declaring as GLOBAL the most frequently used images and screens in an application.

Infrequently used images and screens declared as nonglobal do not put pressure on the buffer pool, because there would not be many copies in the buffer pool for many users, and little I/O with CCATEMP. If an application has many images and/or screens, GTBL might overflow if all the images and/or screens are defined as global.

Consistency checks performed

When a global image or screen is first referred to in a request, GTBL is searched for the object.

If the object is *not* already present in GTBL, the object is written into GTBL.

If the object is present, two consistency checks are performed:

- Persistence—PERMANENT or TEMPORARY status—of the new object must match the persistence of the object already in GTBL.
- Definition—PERMANENT or TEMPORARY—of the new object must match the definition of the object already in GTBL.

Note: If a global image has an array that uses the DEPENDING ON %*variable* option, the names of the %variables are not compared when the definition check is performed.

If either of these consistency checks fails—for example, an image is declared as TEMPORARY, but the same image already exists in GTBL, declared as PERMANENT—then the request is canceled and the following message is generated:

```
M204.2158: GLOBAL object-type DEFINITION DOESN'T
MATCH DEFINITION IN GTBL error-description, NAME
HASH=hash-code
```

Using global images and screens

The following example illustrates using global images and screens to pass screen and image data between User Language requests. The example consists of the following procedures:

- SCREENDEF defines a global screen.
- IMAGEDEF defines a global image.
- PROCA sets the screen item value and an image item value.
- PROCB displays the updated image item and the original screen item.

```
PROCEDURE SCREENDEF
   * KEEP GLOBAL SCREENS AND IMAGES IN SEPARATE PROCEDURES
   * FOR CENTRAL DEFINITION
   SCREEN SCREEN1 GLOBAL
      PROMPT 'SCREEN1'
      INPUT ITEM NUMERIC LEN 4 AT 10
   END SCREEN
END PROCEDURE

PROCEDURE IMAGEDEF
   * KEEP GLOBAL SCREENS AND IMAGES IN SEPARATE PROCEDURES
   * FOR CENTRAL DEFINITION
   IMAGE IMAGE1 GLOBAL
      ITEM IS FLOAT LEN 8
   END IMAGE
END PROCEDURE

PROCEDURE PROCA
   BEGIN
   INCLUDE SCREENDEF
   INCLUDE IMAGEDEF
   * INITIALIZE SCREEN1 AND IMAGE1
   PREPARE SCREEN SCREEN1
   PREPARE IMAGE IMAGE1
   * USER SEES SCREEN1 AND ENTERS A VALUE N (E.G., 10) AS INPUT
   READ SCREEN SCREEN1
   %IMAGE1:ITEM = %SCREEN1:ITEM + 3
   END
END PROCEDURE

PROCEDURE PROCB
   BEGIN
   INCLUDE SCREENDEF
   INCLUDE IMAGEDEF
   * DO NOT INITIALIZE, JUST IDENTIFY FOR SUBSEQUENT REFERRAL
   IDENTIFY IMAGE IMAGE1
      * USER SEES UPDATED IMAGE ITEM (N+3) IN NEW REQUEST
   PRINT 'IMAGE ITEM = ' WITH %IMAGE1:ITEM
   * USER SEES ORIGINAL SCREEN ITEM WITH DATA THAT USER ENTERED
   READ SCREEN SCREEN1
   END
```

```
END PROCEDURE
```

Note: You can run several procedures between PROCA and PROCB with the same results as if you ran those two procedures sequentially, because the defined global objects persist.

System administration issues

To review the implications of the global images and screens feature for system administrators, see the *Model 204 System Manager's Guide* on the following topics:

- GTBL size
- VTBL usage with COMMON images and screens
- User since-last statistics

Clearing the GTBL work area

When you log off, all global objects and global variables are cleared. If you want to clear some or all global objects or global variables from GTBL during your working session, prior to logging out, you can issue:

- CLEAR statement
- CLEARG and/or CLEARGO commands
- $DELG function to selectively delete global string variables
- RESET command for the GTBLEHASH and GTBLPCT parameters
- UTABLE command that changes the size of FTBL, XTBL, or GTBL clears all global objects of any type

Using the CLEARG and CLEARGO commands

Use this command	To Remove…
CLEARG	Only global string variables.
CLEARGO	*All* global objects: images, screens, menus, found sets, lists, and temporary and permanent positions from GTBL; it does *not* clear global string variables.

The CLEARGO command takes no arguments. See the *Model 204 Command Reference Manual* for syntax and detailed information on the CLEARG and CLEARGO commands.

Using the $DELG function

You can use the $DELG function to delete global string variables created by either $SETG or $INCRG.

For syntax and usage of the $DELG function, see “$DELG” on page 27-32 in the *Model 204 User Language Manual.*

The $DELG function deletes a single or group of similar global string variables, releasing and compacting the GTBL space for reuse.

Using the CLEAR statement

You can use the CLEAR statement to clear global objects of the same type, an individual global object, or all global string variables, as shown in the following table:

The statement	Clears…
CLEAR *[type-of-object]* OBJECTS	A class of global objects or all global objects. Global string variables are not cleared, because they are not objects.
CLEAR GLOBAL	Individual global objects from GTBL.
CLEAR GLOBALS	All global string variables. You cannot selectively clear individual global string variables with the CLEAR statement.

Syntax The format for the CLEAR statement is:

```
CLEAR {[[ALL | TEMP | LISTFDST | POSITION] [GLOBAL]]
  OBJECTS
  | GLOBAL {IMAGE | SCREEN | MENU | LIST | FOUNDSET
    | POSITION [PERM | TEMP]} {'objectname'
    | %variable}}
  | GLOBALS
```

Where

- *ALL* clears all permanent and temporary global objects, including: found sets, lists, images, menus, positions, screens, and sorted sets from GTBL.
- *TEMP* clears only global objects explicitly declared as TEMP in your request, including images, menus, and screens from GTBL.
- *LISTFDST* clears all global found sets, lists, and sorted sets from GTBL.
- *POSITION* clears all permanent and temporary positions from GTBL.
- *OBJECTS* specifies CLEAR command is operating on the object type you specified or all types of objects.
- *GLOBAL* clears a specific global object from GTBL. You cannot clear a global string variable created with the $SETG function using this form.
- *objectname* is the literal name of the specific global object to be cleared from GTBL. Enclose *objectname* in single quotation marks.
- *%variable* contains a value that specifies the global object to be cleared from GTBL.
- *GLOBALS* clears all global string variables created with the $SETG function from GTBL. This form does not clear any global objects.

Usage notes The following statements clear all permanent and temporary global found sets, images, lists, menus, positions, screens, and sorted sets:

```
CLEAR OBJECTS

CLEAR ALL OBJECTS

CLEAR ALL GLOBAL OBJECTS

CLEAR GLOBAL OBJECTS
```

Clearing global found sets and lists

You can clear global found sets and lists in GTBL without logging off using the following examples:

- Clear a specific global list or found set by issuing one of these statements:

  ```
  CLEAR GLOBAL LIST 'objectname'

  CLEAR GLOBAL LIST %variable

  CLEAR GLOBAL FOUNDSET 'objectname'

  CLEAR GLOBAL FOUNDSET %variable
  ```

- Clear all global found sets and lists with the statement:

  ```
  CLEAR LISTFDST [GLOBAL] OBJECTS
  ```

  (Global found sets and lists are among the global objects cleared by the CLEARGO command.)

- Close a file or group with which a global list or found set is associated. This includes the file close processing done when exiting or stopping a subsystem.

- Issue a UTABLE command that changes the size of FTBL, XTBL, or GTBL.

If you clear a global list or global found set, then any request that uses the global and does not have the FIND statement that creates the global object receives either of the following error messages:

```
M204.0301 REFERENCED STATEMENT LABEL UNDEFINED

M204.0311 UNACCEPTABLE STATEMENT REFERENCE
```

RELEASE and COMMIT RELEASE statements with global foundsets and lists

The RELEASE statements and the COMMIT RELEASE statements empty the contents of a global found set, global sort set, or global list. The label and positions associated with a found set, sort set, or the list is still considered global, but it is empty. Global positions are not cleared by RELEASE or

COMMIT RELEASE statements, however, without records there is nothing to process.

Clearing remembered positions

Each REMEMBER statement creates a GTBL entry, whether or not GLOBAL is specified. If GLOBAL is not specified, the position is temporary and is cleared from GTBL at the end of request execution.

All remembered positions, along with all other GTBL entries, are cleared when you log off. You can also clear remembered positions in the following ways:

- You can clear a specific temporary position by issuing one of these statements:

  ```
  CLEAR GLOBAL POSITION TEMP 'objectname'
  ```

  ```
  CLEAR GLOBAL POSITION TEMP %variable
  ```

- You can clear a specific permanent position by issuing one of these statements:

  ```
  CLEAR GLOBAL POSITION PERM 'objectname'
  ```

  ```
  CLEAR GLOBAL POSITION PERM %variable
  ```

- You clear all global found sets, lists, or sorted sets with which a remembered position is associated.

  ```
  CLEAR LISTFDST [GLOBAL] OBJECTS
  ```

- You clear all remembered positions by issuing the statement:

  ```
  CLEAR POSITION [GLOBAL] OBJECTS
  ```

Usage notes When you are using the CLEAR GLOBAL POSITION statement, PERM is the default.

The CLEARGO command has the effect of clearing all remembered positions.

Any UTABLE command that changes the size of FTBL, GTBL, or XTBL clears all remembered positions.

Timing and placement of a CLEAR statement

The CLEAR GLOBAL statement takes effect at evaluation time and its placement in a request can have significant consequences:

- If the global object being cleared is referenced in the same request, it is marked as pending clear. The global is then emptied (zero records), and all its record locks are removed. It is cleared and deleted from GTBL at the end of the request.

- If the global object is not referenced in the same request, it is cleared immediately when the CLEAR GLOBAL statement is executed.

Any list, found set, or sort set that is cleared by a CLEAR GLOBAL statement or CLEAR command is no longer GLOBAL and is not available to any subsequent request.

If you specify the CLEAR GLOBAL statement for an object that is not declared as a global, your request is canceled.

Performance considerations

When an entry is cleared from GTBL, other entries in its area are moved as necessary so that all the free space remains between the two main areas. Therefore, when you are clearing more than one entry from GTBL, it is most efficient to begin by clearing the entry that is stored nearest the free space.

For example, if you want to clear all temporary and permanent global images, it is best to clear the temporary images first.

Similarly, if you are clearing two entries within the same category, it is more efficient to clear them in the reverse of the order in which they were created.

The forms CLEAR ALL, CLEAR TEMP, CLEAR LISTFDST, and CLEAR POSITION present exceptions to this principle, because they do not move any data around in GTBL.

21 Large Request Considerations

In this chapter

Overview

There is a practical limit to the number of statements allowed in each request. This limit varies depending on the installation and the type of statements used.

Requests that are too large exceed the amount of memory allotted for one or more of the internal work areas, also referred to as server tables. Such an event produces an error message of the form:

```
REQUEST TOO LONG - xTBL
```

You can take a combination of any of these approaches to handle large requests:

- Rewrite large requests to make them shorter and less complex.
- Increase the sizes of internal work areas (server tables).
- Use the request continuation feature.

This chapter focuses on the second two approaches.

User Language internal work areas

Model 204 uses several internal tables as work areas for processing User Language requests.

The size of these tables (which can be set by the user) determines the number of statements allowed in each request. The number of statements allowed also depends on the types of statements used.

Summary of work areas

The internal work areas required by the User Language compiler and evaluator are listed in Table 21-1:

Table 21-1. User Language internal work areas

Table	Contents
FSCB	Menus, screens, and images
FTBL	File groups
GTBL	Global variables
ITBL	Dummy strings and $READ responses
NTBL	Statement labels, list names, variables
QTBL	Statements in internal form
STBL	Character strings
TTBL	List of temporary work pages
VTBL	Compiler variables

Each of these tables is described in more detail in “Description of tables” on page 21-5.

Resetting table sizes

The size of these tables can be reset by the user with the UTABLE command. Table sizes are controlled by the LxTBL parameters described in the *Model 204 Command Reference Manual*. Resetting these table sizes can increase the total server size requirement at your site. If this is the case, you must refine your request using the techniques discussed in this chapter.

The size of the FSCB is controlled by the LFSCB parameter.

Pushdown list and QTBL size increase

Note that if you get unrecoverable errors when running a request, you might need to increase the size of the pushdown lists or QTBL parameters or both.

Use the UTABLE command (refer to the *Model 204 Command Reference Manual*) or see your system administrator to increase and reset these tables:

- The pushdown list parameter, LPDLST, approximately ten to twenty percent. (Note that, in Version 3.2, the pattern matcher work space uses LPDLST instead of LCPDLST. See “Pattern matching” on page 4-20 for more information on pattern matching.)
- The server pushdown list parameter, LSERVPD, approximately ten percent.
- The LQTBL approximately one percent.

These percentages might vary at your site.

Description of tables

Each table and its use by the User Language compiler and evaluator is described separately below. The descriptions should help you to restructure large requests to make more efficient use of the available table space.

FSCB (full-screen buffer)

The FSCB (full-screen buffer) is used to store menu, screen, and image definitions and the values of screen variables and image data blocks. FSCB space is reused by each logical menu definition, logical screen definition, or block definition. Therefore the FSCB need only be large enough to hold the largest menu, screen, or image definition. FSCB requirements are:

- Every menu requires 144 bytes of fixed overhead in the FSCB, which includes the menu title. In addition, each menu prompt adds another 144 bytes in the FSCB.
- Every screen requires 432 bytes of fixed overhead for the first panel and 144 bytes (includes the screen title) for any other panels. An additional 32 bytes is used for each screen prompt and input item, as well as a 32-byte entry for every screen line containing at least one input item. Each defined screen line adds 80 bytes to FSCB requirements, including skipped lines.

  Additional space is used in the FSCB if automatic validation options are used. Each validation option adds 2 or 4 bytes to the FSCB. In addition, VERIFY adds 256 bytes the first time a particular character set is used in a logical panel. Additional occurrences of the same character set do not add extra space. ONEOF and character RANGE store each character string plus one byte for each string’s length. NUMERIC RANGE adds 8 bytes for each number (16 bytes for each range pair).
- Every block used in image definition requires space in the FSCB. The amount of space required is computed as the sum of the following:
  - 32 bytes
  - 16 bytes for each IMAGE statement
  - 32 bytes for each ARRAY statement
  - 16 bytes for each item definition
  - 16 bytes for each OCCURS clause in an item definition
  - 32 bytes for each INITIAL clause in an item definition that specifies a value or number (other than the ZERO keyword) plus the length of the value or number
  - Total length of the block in bytes

FTBL (file group table)

Data structures relating to file groups are stored in FTBL.

Two types of FTBL entries are allocated by Model 204 when file groups are used:

- The first type is allocated each time a group is opened (explicitly by OPEN or implicitly for an ad hoc group) and is released when the group is closed. This type of entry has a fixed-size portion of 62 bytes, plus 2 bytes for each file in the group definition.
- The second type is used for collecting field name codes and properties during a request. An entry is allocated each time a new field name is encountered in the request. The entry size is variable, consisting of 9 fixed bytes plus a number of bytes equal to the length of the field name plus 11 bytes for each file in the group. These field entries are deleted at the END statement (including END MORE).

In addition to the space required by the two types of entries described above, Model 204 allocates a fixed amount of space in FTBL equal to 4 bytes times the value of the NGROUP runtime parameter.

GTBL (global variable table)

Global variables, lists, found sets, images, screens, menus, and REMEMBERed positions in FOR loops are special mechanisms for communicating across User Language request boundaries, as described in Chapter 20.

A GTBL entry is created when any global object is declared. In addition to the entries created by individual declarations, the last 32 bytes of your GTBL space allocation is reserved for a GTBL trailer.

You can delete entries from GTBL by using different forms of the CLEAR statement, or the CLEARG or CLEARGO commands. See “Clearing the GTBL work area” on page 20-28 for detailed instructions on clearing objects.

The types of GTBL entries are listed below:

- A global variable (declared or changed with the $SETG or $INCRG function) creates an entry with a length of 4 bytes plus the length of the variable name plus the length of the current value. If a global variable is redefined, its old entry is deleted and a new entry is added.
- A global list or found set in single file context creates an entry with a length of 28 bytes.
- A global list or found set in group context creates an entry with a length of 28 bytes plus eight bytes for each file in the group.
- A sorted found set creates an entry with a length of 40 bytes.
- A position in an unordered or sorted record set creates an entry with a base length of 32 bytes.

- A position in a non-pattern driven ordered (BTREE) record set creates an entry with a length of 60 bytes plus a number of bytes equal to the size of the BTREE key.
- A position in a pattern driven ordered (BTREE) record set creates an entry with a length of 68 bytes.
- A position in a sorted file record set creates an entry with a length of 80 byes.

ITBL (dummy string and $READ response table)

ITBL is used to hold dummy string and $READ responses entered as arguments to an INCLUDE statement or command.

Argument strings are saved as entered, including delimiters, with an additional 4 bytes of overhead for each saved string.

Space taken by a string is released when the included procedure is exhausted.

NTBL (statement labels/list names/variables table)

One entry (12 bytes) is allocated in NTBL for each statement label, list name, %variable, and image, menu, and screen variable. An entry also is allocated for each partner process opened by a request. Data defined as common takes two entries for the first COMMON declaration and one additional entry for each additional COMMON declaration.

One entry is allocated for each unlabeled FIND. An extra entry is allocated for each FOR EACH VALUE statement or each FOR statement with the IN ORDER clause. Most NTBL entries are preserved by MORE, but the unlabeled FIND and secondary FOR entries are deleted.

During request evaluation, one entry is required for each sequential or VSAM file opened simultaneously.

QTBL (internal statement table)

Each statement is compiled into internal Model 204 instructions in QTBL. After compilation, the entries in QTBL are used to drive the evaluator. The entries generated by each User Language statement vary in number and in size.

Under certain circumstances, users sharing precompiled procedures can use a shared version of QTBL, thus reducing server I/O. See the *Model 204 System Manager's Guide* for more information.

Some common examples are listed on the following pages.

Statement or other functions	**Bytes generated**	**Comments**
ADD	20	Each *fieldname=value*
CALL	16	
CHANGE	44	*fieldname=value* TO
CLEAR ON	16	
CLEAR TAG	16	
CLOSE	16	
CLOSE PROCESS	16	
COMMIT	4	
COMMIT RELEASE	20	
COUNT RECORDS	52	Plus 20 or a group
DELETE	24	fieldname=vaue
DELETE RECORD	16	Each
END	4	
Expressions	Varies	Depends on the complexity of the expression and on whether conversion between data types is required.
FIND ALL RECORDS (with no qualification)	64	• Record security adds 52 bytes. • Group adds 20 more. • Each inverted condition adds about 36 bytes; NUMERIC RANGE conditions a little less. • Each direct and Ordered Index condition (for example, AFTER, GREATER THAN) generates an entry of 20 bytes in length. • The IN RANGE FROM and TO clause and BETWEEN operator clause generate entries of 28 bytes, plus 16 for each AND (except where AND is part of the BETWEEN clause). • An additional 16 bytes are added for each FIND statement with at least one direct condition. • AND and OR as operators generate 16 bytes. • At request evaluation time, additional space can be required in QTBL for a FIND with direct conditions; this requirement never exceeds the number of direct bytes already compiled by more than 16 bytes. The extra QTBL space is released after evaluation of the FIND. Numeric conditions, when used with NUMERIC RANGE fields, also generate bytes during evaluation that are released as soon as the FIND has been executed.
FIND ALL VALUES	• 32 • 74	• For an ORDERED field • For an FRV field

Statement or other functions	**Bytes generated**	**Comments**
FOR EACH RECORD	20	In addition to the body of the loop.
FOR EACH VALUE OF	88	In addition to the body of the loop: • IN ORDER adds 68 bytes • A group adds 20 more
Function call	16	Plus the bytes necessary to evaluate the arguments.
IDENTIFY	16	
IF	32 for each line in the IF statement	• THEN clause generates 16 bytes, plus additional for the body of the clause. • Each operator in the IF statement generates 16 bytes. • Conversions between strings and numbers take 16 bytes each. • Each ELSE and ELSEIF generates 16 bytes, plus additional for the body of the ELSE or ELSEIF.
Index loop	40	In addition to the bytes necessary to evaluate expressions used in the statement, plus any generated within the body of the loop.
INSERT	24	Each *fieldname=value*
MODIFY	16	
NOTE	20	Each *fieldname=value*
ON units	16	Each ON unit also generates bytes for the statements within the ON unit and 16 bytes for the automatically generated STOP statement.
OPEN	16	For external entities.
OPEN PROCESS	16	
PAUSE	16	
POSITION	20	
PREPARE IMAGE	8	
PREPARE MENU	8	
PREPARE SCREEN	8	
PRINT	16	For each term in the print specification: • Field from a record takes an additional 20 bytes. • AND connective generates 16 bytes. • TAB generates 4 bytes. • WITH generates no bytes.
PRINT MENU	16	
PRINT SCREEN	16	

Statement or other functions	Bytes generated	Comments
READ IMAGE	16	
READ MENU	20	
READ SCREEN	20	
RECEIVE	16	
References to a subscripted %variable	16	in addition to the bytes required to evaluate the expression for each dimension.
RELEASE POSITION	8	
REPEAT	16	Additional bytes are required to evaluate each condition for the WHILE clause.
REREAD SCREEN	20	
RETRY	16	Each pending statement.
RETURN	16	
SEND	16	
SIGNAL PROCESS	12	
STORE RECORD	16	Plus 16 for each field to be included in the record.
TAG	16	
TRANSFER	16	
WRITE IMAGE	12	

QTBL is emptied by END and END MORE (see “Starting and ending requests and continuations” on page 21-17 for a discussion of END MORE).

STBL (character string table)

All character strings are stored in STBL. The strings are kept in counted form, with a 1-byte length preceding the string itself.

The following types of strings are stored in STBL:

- Quoted strings
- Values in FIND specifications
- Literal values in ADD, CHANGE, DELETE, and INSERT statements
- %Variable values (space reserved for maximum length) for STRING %variables

- Array %variables

During evaluation, the following types of strings also are stored in STBL:

- Results of functions that return character strings
- Intermediate string results during arithmetic expression evaluation
- NOTE values and fields to be printed
- FOR EACH VALUE values
- FOR EACH OCCURRENCE values
- FOR EACH RECORD IN ORDER BY values when the Ordered Index is used to drive the loop

The space utilized for the storage of intermediate results during the evaluation of an arithmetic expression is freed when the evaluation of that expression is completed.

The last three categories above occur only within FOR EACH RECORD, FOR EACH OCCURRENCE, or FOR EACH VALUE loops. Each pass through the loop reuses the STBL space from the previous pass. When the loop is finished, the last NOTE, FOR EACH OCCURRENCE, or FOR EACH VALUE values remain in STBL (the space is not freed). Thus, if FOR statements are executed a large number of times or if many NOTE statements are issued, STBL fills up rather quickly.

A FIXED or FLOAT %variable array uses 8 bytes of STBL space for each element. If the FIELD SAVE option is used when a STRING %variable array is declared (if field information is saved where the element is used for a field name variable), then 13 bytes plus the maximum length of the string plus 1 byte are reserved in STBL for each element of the array. If the NO FIELD SAVE option is specified, the extra 13 bytes for field information are not reserved. Using NO FIELD SAVE can result in a significant saving in a multi-dimension array. For example, in an array with dimensions 100 by 2, specifying NO FIELD SAVE saves 100*2*13 (or 2600) bytes.

The MORE command (see “MORE command” on page 21-15) releases most STBL space, keeping only the entries used for %variables and arrays. STBL is filled in during evaluation, when a value is assigned to the %variable. The space in STBL is reused when the %variable is reassigned.

TTBL (temporary work page list table)

TTBL entries keep track of scratch file (CCATEMP) pages. The FIND statement uses scratch pages as a work space for evaluating Boolean expressions. The number of TTBL entries depends on the complexity of the Boolean operation. TTBL entries are released at the end of the evaluation of the FIND statement.

VTBL, the compiler variable table

Entries in VTBL vary in size. Most range from 8 to 20 bytes; some are much larger. Many User Language statements and some constructs cause one or more compiler variables to be allocated in VTBL. Common examples are listed in the following sections.

Expressions, commands, and statements

- Arithmetic expressions allocate 8-byte entries for string expressions and 16-byte entries for numeric expressions, some of which are used for intermediate results. The intermediate result space is reused by subsequent expressions.
- COUNT statement allocates one 8-byte entry.
- SORT statement allocates a 12-byte entry, plus one 20-byte entry for each field referenced in the sorted records (or one entry for a PRINT ALL INFORMATION statement), plus one 28-byte entry for each sort key.
- User Language function calls allocate one entry that is 4 + (4*number-of-arguments) bytes long. The arguments themselves have their own VTBL entries that are independent of the function call.
- Each subroutine declaration takes 16 bytes, plus the associated space for %variables and lists used as parameters.
- FIND allocates one basic 8-byte entry for a single file or 8+(8*number-of-files) bytes for a group. At least two 20-byte entries and one 28-byte entry are allocated for work space (more for complex Boolean operations). Also, one entry is allocated for each fieldname = value (property) pair mentioned in the FIND. The length of a property entry is at least 20 bytes; more for large files. Each direct retrieval requires an additional 28 bytes in VTBL. All space except that allocated by the basic entry is released after the FIND has been evaluated and is reused by subsequent FINDs.

  Each Ordered Index retrieval requires an entry of 4 bytes plus 4 bytes per segment in the file.

Defined objects

- Starting in Version 5.1, the size of the header record of a value set is increased from 20 bytes to 48 bytes.
- Each %variable adds one entry. The entry is 16 bytes for a FIXED %variable, 12 bytes for a FLOAT %variable, and 16 bytes for a STRING %variable. Each %variable array adds one 24-byte entry. There is also a 12-byte entry for every reference to an array element, regardless of the number of dimensions in the subscript.

- Every menu definition adds a 48-byte entry. Every screen definition adds a 68-byte entry, plus 4 bytes for each physical screen panel. A one-panel screen entry in VTBL is 72 bytes.
- Every set of related images adds a 12-byte entry plus 4 bytes for approximately every 256 items in the block.
- Lists allocate 8-byte entries for single files or 8+(8*number-of-files) bytes for groups.

CALL statements

- Each CALL statement with parameters allocates one entry that is 4 + (4 * number of arguments) bytes in length.
- Evaluation of a CALL statement generates one 28-byte entry that is released by RETURN. These entries are placed in a LIFO stack area.

Value loops

- FOR EACH RECORD without the IN ORDER clause allocates a 16-byte entry.
- FOR EACH VALUE with FROM, TO or LIKE specified, and/or using an ORDERED field allocates a 40-byte entry.
- FOR EACH VALUE and FOR EACH RECORD IN ORDER BY, using an ORDERED field, allocate an additional 44-byte entry.
- Every FOR loop position declared with a REMEMBER statement adds an 8-byte entry.
- When an ON unit is invoked, one 28-byte entry is used. The entry is released by BYPASS, RETRY, or STOP.

Many VTBL entries are deleted by the MORE command (refer to “MORE command” on page 21-15). Entries are retained in the following cases:

- %Variables
- COUNT’s 8-byte entry (Not retained if the original User Language statement was unlabeled.)
- FIND’s basic 8-byte (larger for groups) entry (Not retained if the original User Language statement was unlabeled.)
- Entries for lists
- Image, menu, and screen
- SORT’s basic 12-byte entry plus the 20-byte sort key entries. (Not retained if the original User Language statement was unlabeled.)

Entries in the last three categories are not retained if the original User Language statement was not labeled.

Request continuation

A ***continuation*** is a request that refers to certain items in the preceding request.

Request continuation is used to increase user interaction with Model 204 without increasing retrieval costs, or to break up into smaller logical units requests that would be too large to run otherwise (requests that would exceed work table space, as described in the previous sections).

Request continuation is particularly effective when the request consists of a large and complex retrieval section followed by a similarly complicated report generation section.

Specific benefits of request continuation include:

- The opportunity to enter certain Model 204 commands between a request terminated by END MORE and its continuation.
- The ability to decide whether to continue a request based on its (intermediate) results.
- The opportunity to reduce the size of your QTBL, since this work area is cleared with the execution of an END MORE statement (unlike other work areas, which retain their contents when END MORE is encountered).

See the next section for rules relating to the use of request continuation.

MORE command

Description

If a BEGIN command is replaced by the MORE command, the request that follows is considered to be a continuation of the previous request and can refer to statement labels and other information from that request.

Example 1

Suppose the user is interested in high vehicle premiums for a particular garaging location. The user might write the following request:

```
BEGIN
HIGH.PREM: FIND ALL RECORDS FOR WHICH
               VEHICLE PREMIUM IS GREATER THAN 200
               GARAGING LOCATION = VA03
           END FIND
END MORE
```

Before printing information for the record set, the user can determine the amount of records to be printed by adding the following statements:

```
MORE
LOW.DEDUCT: FIND AND PRINT COUNT
                FIND$ HIGH.PREM
            END FIND
END MORE
```

The LOW.DEDUCT statement refers to the records retrieved by the HIGH.PREM statement in the original request. In this way, the expense of retrieving that set of records a second time is avoided.

If this continuation indicates that there are less than 40 records meeting the extended retrieval conditions, the user can continue the request again:

```
MORE
            FOR EACH RECORD IN LOW.DEDUCT
                PRINT OWNER POLICY -
                     WITH PRINCIPLE DRIVER TO COLUMN 25
            END FOR
END
```

Example 2 In the preceding example, the user also could redirect the output, depending on the number of records found. For example, if there were more than 100 records in the set the user could precede the request continuation with a USE command:

```
USE OUTPRINT
MORE
            FOR EACH RECORD IN LOW.DEDUCT
                PRINT OWNER POLICY -
                     WITH PRINCIPLE DRIVER TO COLUMN 25
            END FOR
END
```

In this way the user can decrease the output time without having to reduce the number of records found in the initial request.

Rules for request continuation

Avoid too many continuations

In theory, requests can be continued indefinitely. However, information saved from the pieces of a basic request can fill the internal work areas (see “User Language internal work areas” on page 21-3), preventing further continuations.

Sets of records retrieved in one request remain locked through all continuations.

For these reasons, you should issue a fresh BEGIN command when information from the last request is no longer needed.

Starting and ending requests and continuations

A request is started by a BEGIN command; a continuation is started by a MORE command. Both are terminated by END or END MORE statements (with a USE specification if desired).

Multiple continuations

You can continue a request several times, as long as you issue an END MORE statement at the end of each section to be continued. Each new continuation, the initial request, and all previous continuations constitute the basic request. Once a request or a continuation ends with an END statement without the MORE option, it is no longer continued.

References in a continuation

A continuation can refer to the following elements of the basic request:

- Statement labels for COUNT, FIND, and SORT statements
- List names
- %Variables
- Menus and screens
- Images

Statement labels for statements other than COUNT, FIND, and SORT in the basic request can be reused in the continuation.

When a request is continued, all found sets, lists, and %variables of the main request are preserved for later use. The found sets, lists, and %variables of complex subroutines are discarded after END MORE unless they are common.

If a request opens a dataset, and you want to leave the dataset open after the request terminates, you can end the request with END MORE USE. This allows the continuation to add to the USE output without resetting the page number.

Restrictions applying to request continuations

A request continuation *cannot*:

- JUMP TO statement labels or CALL subroutines in the basic request
- Alter the type, length, or decimal place parameters of any %variable mentioned in the basic request
- Use a FOR RECORD NUMBER IN statement that refers to a label in the basic request

A basic request cannot continue if the user issues any of the following commands:

- BEGIN (a new request is started)
- CLOSE
- FILELOAD
- FLOD
- INITIALIZE
- LOGIN
- LOGOUT
- MONITOR
- UTABLE

ON units

An ON unit definition is not preserved across an END MORE statement and MORE command. Each new request continuation must define its own ON units.

Interaction with SORT statement

The records produced by a SORT statement contain only fields necessary to satisfy the initial request (or continuation) in which the SORT statement appears. If the initial request contains these statements:

```
SORT.RECS: SORT RECORDS IN FIND.RECS BY FULLNAME
           FOR EACH RECORD IN SORT.RECS
               %A = AGENT
```

a continuation retrieves only the AGENT field from the sorted records. All other fields are treated as missing in the sorted records.

Part IV Application Development

Part IV describes two ways to develop Model 204 applications.

Using the full-screen feature you can design menus and screens for displaying data and accepting end user input.

The Subsystem Management facility provides a more sophisticated way of creating end user applications. Subsystem Management features include:

- Minimized end-user intervention
- Improved performance through precompiled procedures
- Centralized error handling
- Security facilities

22

Full-Screen Feature

In this chapter

- PRINT MENU statement
- MODIFY and PREPARE MENU statements
- Menu manipulation

Screen definition and manipulation topics

- Defining screens
- SCREEN and END SCREEN statements
- TITLE and PROMPT statements for screens
- INPUT statement
- Automatic validation options for INPUT
- DEFAULT statements
- SKIP and NEW PAGE statements
- MAX PFKEY statement for screens
- INCLUDE statement
- Screen definition example
- Screen manipulation
- READ SCREEN statement
- REREAD SCREEN statement
- TAG and CLEAR TAG statements
- Cursor handling
- MODIFY and PREPARE statements for screens
- Cursor handling
- READ, REREAD, and PRINT evaluation sequence
- Screen manipulation example

Use of the Screen/Menu feature with Line-at-a-Time terminals

- Line-at-a-time terminal support

Overview

The Model 204 full-screen feature provides data entry capability invoked through User Language. The full-screen feature allows the entire screen of a video display terminal (an IBM 3270) to be formatted, displayed, and accepted as a single entity, rather than field by field.

User Language also supports line-at-a-time screen handling, which is described in this chapter.

Menus and screens

The full-screen feature provides two major capabilities:

Capability	Displays…
Menu	Options or programs that can be selected by a terminal operator.
Screen	Information for inquiry or updating purposes. Screens frequently are used for data entry applications in which a terminal operator views a formatted screen and enters data in response to specific prompts. The application then validates the entered data, flags errors, and prompts for corrections.

You can define and manipulate each display type in a User Language request.

Most statements used with screens also apply to menus, with some minor differences. The first part of this chapter discusses menu definition and manipulation, and the second part discusses screen definition and manipulation.

LFSCB parameter setting

The system manager sets the LFSCB parameter (size of full-screen buffer) to a positive number during system initialization, or the user sets the LFSCB parameter Online with the UTABLE command. For details regarding the LFSCB parameter, refer to the *Model 204 Command Reference Manual*.

Maximum number of screens and menus

The maximum number of screens and menus combined is 256 per request.

Global screens and menus

Global screens and menus provide a means for passing screen and menu data from one request to another, and for efficiently managing more than one screen or menu in one request.

Although there are some differences in the way you use the DECLARE SCREEN and DECLARE MENU statements, you generally define global screens and menus in the same way as non-global screens and menus. See “Global images and screens” on page 20-21 for a discussion.

This chapter describes these statements for global and non-global screens:

- DECLARE MENU
- DECLARE SCREEN
- MODIFY
- PREPARE

Screen and menu formatting

The full-screen feature specifies:

- Where screen items are to appear on the video display terminal
- How screen items are displayed (for example, highlighted and/or in color)

Screen and menu items

Screens and menus can include:

- Title (name of the screen or menu)
- Prompts
- Input areas (for screens only)

Screen and menu definition

Screen and menu definitions begin with a SCREEN or MENU statement, respectively, and end with a corresponding END SCREEN or END MENU statement. The screen or menu definition is made up a series of statements that define the screen/menu components: title, prompt(s), and input area(s).

Screen and menu manipulation

User Language provides a number of statements that read previously defined screens and then accepting input items from the terminal operator. You can also use screen manipulation statements to redisplay corrected screens, restore previous default values for screen items, and flag incorrect values.

Full-screen variables

Full-screen application design normally requires the use of special variables:

- Menu and screen variables—Refer to titles, prompts, and input items.

- Reserved variables—Have special meaning to Model 204; used to perform special functions.
- Screen item name variables—Refer indirectly to screen items.

Full-screen processing

To design data entry applications using the full-screen feature, you must accommodate the sequence of events that normally occurs during a data entry session.

Menu displayed

The application request normally begins the application by displaying a menu to the terminal operator. A menu typically contains a title and formatted prompting information. For example:

```
    EMPLOYEE MENU

1  ADD EMPLOYEE
2  CHANGE ADDRESS
3  CHANGE INSURANCE
4  ADD DEPENDENT
```

Operator interaction with menu

The operator (application end user) selects a menu option by indicating the appropriate option number with a Program Function (PF) key or by tabbing to the desired selection number (using →, ←, or ↵) and pressing the ENTER key. Control is then transferred to the selected option's program. The operator is reprompted if the cursor is not positioned correctly.

Screen displayed

The screen for the selected option is then displayed. A screen typically contains a title, formatted prompting information, and areas in which the user can enter data. For example:

```
      EMPLOYEE ADD SCREEN

 FILL IN THE FOLLOWING INFORMATION
 FOR EACH NEW EMPLOYEE

 NAME:
 STREET:
 CITY:
 STATE:
 ZIP:
 AGE:
 SEX:
 SSN:
```

In this example, each prompt is followed by an input area in which data can be entered. Titles and prompts are protected and cannot be modified by the operator.

The operator presses the tab key on the terminal to move between input areas, filling in data. The operator can tab backward or forward on the screen, entering data. The input areas can be filled in any order, and can be corrected if erroneous data is entered, as long as the screen has not been transmitted. When data entry for the screen is completed, the operator presses the ENTER key or a PF key to transmit the data to Model 204.

Input validation

The input entered at the terminal is then validated according to criteria specified in the application request. Two types of validation can be performed:

Type of validation	Request can specify…
Automatic	Validation criteria (for example, the response must be all-numeric) using special full-screen options.
Manual	Statements that check for errors in terminal responses (for example, statements that verify that an input code matches a code in a particular record).

Items that do not pass the validation criteria are tagged with an error indicator. If the screen is redisplayed for correction, the error indicator appears in column 80 of the line containing the error. An example of a screen redisplayed for correction is shown below.

```
     EMPLOYEE ADD SCREEN

REENTER VALUES MARKED WITH *


NAME: JOE SMITH
STREET: 87 OAK DRIVE
CITY: NORFOLK
STATE: VA
ZIP: 501B3                                  *
AGE:14                                      *
SEX: M
SSN: 042-54-9803
```

The terminal operator can then correct the items in error and press ENTER when all corrections have been made. This cycle is repeated until all items on the screen pass the validation criteria. Control is then returned to the request to continue processing.

Application display considerations

Prior to application design, you should be familiar with the following aspects of the full-screen feature and the terminal display area:

- Positions on the video screen, often described as layout
- Terminal display attributes for color and light intensity

Screen display area

The typical 3270-type video display terminal consists of 24 rows of 80 columns. Every screen position, except the system-reserved positions specified below, can be used for menus and screens.

Position	**Is reserved for…**
Column 1 (the leftmost column)	Internal system use.
Columns 2 through 4	System-generated menu selection numbers when menus are displayed. These columns are not reserved for menu titles or for screen use.
Column 80, the rightmost column	Error indicators when user-defined screens are displayed. This column is not reserved for menu use.
Line 1, the top row on the screen	Screen and menu titles that can be specified by the user.
Last line, the bottom row on the screen, usually row 24	Model 204 backpaging feature. Backpaging allows the terminal operator to review previous pages of output. For detailed information about backpaging, refer to the *Model 204 Terminal User's Guide*.

Display attributes

The following basic display attributes available on 3270-type terminals can be used within a full-screen application design. Abbreviations are capitalized.

```
VISible     UNPROTected   BRIGHT
INVisible   PROTected     DIM
or
INVISible
```

Extended display attributes

In addition, extended display attributes available on some 3278 and 3279-type terminals can be used within an application to alert the terminal operator to areas of the display and error conditions. To use extended attributes, the FSOUTPUT parameter must be set. For details about the FSOUTPUT parameter, refer to the *Model 204 Command Reference Manual*.

Extended attributes are a combination of display attributes selected from the categories listed below. Abbreviations are capitalized. Consult the appropriate hardware support person within your installation to determine if extended attributes are supported.

Highlighting attributes

```
NOUnderSCORE   NOBLINK   NOREVerse
UnderSCORE     BLINK     REVerse
```

Color attributes

```
BLUE    TURQuoise
GREEN   WHITE
PINK    YELLOW
RED
```

How display colors are assigned

3270-type display devices—both actual 3278 and 3279 terminals, and terminal emulators—operate according to a protocol defined by IBM. Display devices operate in either base-color or extended-color mode. Each time a screen is displayed, one of these modes is selected for display.

If Model 204 displays a screen with…	Then 3270 selects…
NO fields with a color attribute set	Base-color mode
ANY field with a color attribute set	Extended-color mode

Base-color mode attribute assignments

In base-color mode, fields have no color explicitly specified. The display color is determined by whether the field is DIM or BRIGHT and whether the field is PROTECTED or UNPROTECTED, as follows:

Color attributes	Color displayed	Default color for…
DIM, PROTECTED	BLUE	TITLE and PROMPT fields
DIM, UNPROTECTED	GREEN	INPUT fields
BRIGHT, PROTECTED	WHITE	
BRIGHT, UNPROTECTED	RED	TAG fields

Extended-color mode attribute assignments

In extended-color mode, each screen field has a color associated with it. If a field does not have a color explicitly specified, the field is displayed, as follows:

Color attributes	Color displayed	Default color for…
DIM	GREEN	TITLE, PROMPT and INPUT fields
BRIGHT	WHITE	TAG fields

Note: The display of a field without a specified color might be altered if a change to extended-color mode is triggered by the modification of some other field in the screen.

Most terminal emulators provide a mechanism to modify color mappings. Any change to the color mapping using this mechanism is local, and is not reported to Model 204.

Consult the following IBM manuals for additional screen display information:

Part no.	IBM Title
GA23-0059	3270 Data Stream Programmer's Reference
GA23-0218	3174 Establishment Controller - Functional Description

Display attribute rules and restrictions

You should be aware of these rules and restrictions when specifying display attributes:

- The INVISIBLE attribute overrides all other display attributes.
- A color attribute overrides the BRIGHT and DIM attributes.
- Color and highlighting attributes can be mixed.
- Color attributes cannot be mixed together.
- Highlighting attributes cannot be mixed together.
- The PROTECTED attribute implies an autoskip. Autoskip causes a screen item to be skipped by the forward and backward tab keys.

Full-screen variables

Types of variables used

Full-screen application design normally requires the use of special types of variables.

Type of variable	Refers to…
Menu and screen	Titles, prompts, and input items
Reserved	Special meaning to Model 204 and are used to perform special functions
Screen item name	Screen items, indirectly

Menu and screen variables

Every input item area on a screen must have a name. Optionally, a menu, screen title, or prompt also can have a name to which a value is assigned before the menu or screen is displayed.The name must be unique within a particular screen or menu definition. However, the same name can be used on multiple screens or menus.

To avoid the confusion of duplicate names for different menus and screens (for example, a NAME item on an EMPLOYEE screen and a NAME item on a CREDITOR screen), you must refer to a screen or menu name using the following formats:

- **Titles and prompts**—Named titles or prompts on menus and screens are referenced in the following format:

  %menuname:*promptname*

  or

  %screenname:*promptname*

  Values for named titles or prompts are specified by an assignment statement of the following format:

  %menuname:*promptname = value*

  or:

  %screenname:*promptname = value*

  For example, this statement:

  ```
  %EMPLOYEE:PNAME = ’ENTER NAME:’
  ```

  assigns the text string, ENTER NAME:, as the value of the prompt PNAME on the EMPLOYEE screen.

- **Input items**—Input items entered on screens are referenced in the following format:

  ```
  %screenname:inputname
  ```

  For example, this statement:

  ```
  %EMPLOYEE:INNAME
  ```

  identifies the input item INNAME entered on the EMPLOYEE screen.

Reserved variables

Model 204 provides the following reserved variables for storing values entered by the terminal operator:

Reserved variable	Stores…
%menuname:SELECTION	Menu selection number entered by a PF key or by tabbing to the desired selection number and pressing ENTER. The value stored in %menuname:SELECTION can be used: • Directly in either an arithmetic expression or a computed JUMP statement • As the target of an assignment statement to specify the initial cursor position for the next menu display.
%screenname:PFKEY	Value of the PF key number entered by the terminal operator for a screen. If the terminal operator presses the ENTER key, a zero is returned in %screenname:PFKEY. You can use the value stored in %screenname:PFKEY to select the next group of statements to be executed. Typically, this is accomplished with a series of IF statements or a computed JUMP statement.

Screen item name variables

Screen item name variables refer indirectly to screen item names, thereby allowing portions of a request, such as subroutines, to be generalized. A screen item name variable is indicated in the following format:

```
:%screen-item-name
```

During the evaluation of a request, a string value can be assigned to a %variable, as illustrated below:

```
%SCRVAR = 'SCREEN:AGE'
```

and that %variable can then be used in statements where screen item names normally appear.

For example, to tag the screen item AGE, enter:

```
TAG :%SCRVAR
```

A screen item name variable and a menu or screen variable are actually the same variable.

Defining menus

Use menu definition statements to define and format menus. Formatting involves specifying a title and all prompts to be displayed on the screen. A menu definition must be included in the User Language request for every menu to be displayed by that request.

Syntax

The statements described below define a menu. Each menu is defined using the following format:

```
MENU menuname
menuline
     .
     .
     .
END MENU
```

Each menu definition must begin with a MENU statement and end with an END MENU statement. Between these statements, you describe the titles, prompts, and blank lines that are to appear on the menu. This is done by specifying a series of menulines that describe the contents of each line of the menu.

The number of menulines must not exceed 23 lines.

Comments and blank lines included in a menu definition (between MENU and END MENU statements) are ignored by Model 204.

Summary of menu definition statements

Menulines are composed of the statements listed below:

Table 22-1. Menu definition statements

Statement	Description
TITLE	Specifies the menu title.
PROMPT	Defines a particular menu option.
SKIP	Skips one or more lines on the screen.
MAX PFKEY	Specifies the maximum PF key value associated with a menu.

MENU and END MENU statements

MENU statement

The MENU statement must be the first statement in a menu definition. MENU signals Model 204 that the statements specified between the MENU and the END MENU statement define a logical menu.

Syntax

The format of the MENU statement is:

```
[DECLARE] MENU menuname [GLOBAL [PERMANENT
  | TEMPORARY]
  | [PERMANENT | TEMPOARY] GLOBAL | COMMON]
```

Where

- *menuname* is a user-defined string from 1 to 255 characters in length. Every menu name defined in a request must be unique. The menu name specified in the MENU statement is assigned to the menu being defined; it subsequently can be referenced in menu manipulation statements.
- *DECLARE* and *COMMON* specify that the menu is a common element in the request. Specifying COMMON (or specifying neither COMMON nor GLOBAL) results in the menu being stored in FSCB and processed as it would have been prior to Version 2.2. Common elements, as well as DECLARE and COMMON, are discussed in detail in “Sharing common elements” on page 12-15.
- *GLOBAL* specifies that the menu is stored in GTBL. Global menus have an implied scope of COMMON (see above).

  Note: Do not specify the GLOBAL attribute for multi-page (logical pages) menus.
- *PERMANENT (PERM)* can only be used with global menus. PERM global menus persist across request boundaries (they are maintained in GTBL even after a request has been terminated).
- *TEMPORARY (TEMP)* can only be used with global menus. TEMP global menus are allocated in GTBL, but are deleted at request termination. An example of using a TEMP global menu would be when you do not need to pass the menu to another request; using TEMP global menus eliminates the need for you to explicitly delete globals that do not need to persist.

END MENU statement

Every MENU statement must terminate with a corresponding END MENU statement, which must be the last statement in a menu definition.

TITLE statement for menus

The TITLE statement specifies a character string that is displayed as the menu title on the first line of the screen.

If a TITLE statement is not defined in a menu definition, a menu title can be specified when the menu is displayed (see the section titled “READ MENU statement” on page 22-24). If no title is specified, this system menu title is displayed:

```
INDICATE NUMBER FOR DESIRED SELECTION
```

Syntax

A TITLE statement must be the first menuline of a menu definition. The format of the TITLE statement is:

```
TITLE {'text' | promptname} [AT [COLUMN] n]
 [TO [COLUMN] m | [LEN m] [DP {k | *}]]
 [DEFAULT 'value']
 [[READ] attributes] [REREAD attributes]
 [PRINT attributes]
```

Where

- *text* is a character string enclosed in single quotes with a maximum length of 79 characters. If the string exceeds 79 characters, a compilation error occurs.
- *promptname* is a menu variable to which a value can be assigned before the menu is displayed with the READ MENU or PRINT MENU statement.

Location options

- Location options (optional) are listed and described in the following table:

Options	Specifies…
AT *n*	Where the title is to begin on the screen.
TO *m*	Column where the title is to end.
LEN *m*	Length of a title beginning at the n specified in the AT option. If a menu variable is used to supply the title, and you do not specify a length, the default length is the remainder of the line.
DP *k or* DP *	Number of decimal places displayed in a title. • DP *k* displays k decimal places for the title. Any additional decimal places are truncated. • DP * displays all decimal places.

- COLUMN keyword (optional). An *n* value of 1 automatically is converted to 2, because a title cannot begin in the reserved first column. If the AT option is not included in the TITLE statement, the title begins at column 2.

If the difference between the AT and TO specifications or the LEN specification is less than the length of the string, the string is truncated. If the specified length exceeds the length of the string, spaces are used to pad the end of the title. Note that the title string is not right-justified at the column indicated by the TO specification. This TITLE statement indicates that the text string within single quotes is to be displayed, beginning in column 13:

```
TITLE 'EMPLOYEE MENU' AT COLUMN 13
```

- DEFAULT lets you provide a default literal value for a named title. The value must be enclosed in single quotes. This option eliminates the need to use an assignment statement to set the initial value of a TITLE variable.
- READ selects the display attributes for the title on execution of a READ statement. The default display attributes for a title are PROTECTED, DIM, and VISIBLE.

  Note: The UNPROTECTED attribute is not allowed in a title. Specifying the UNPROTECTED attribute results in an error message.
- PRINT selects the display attributes for the title on execution of a PRINT statement. If no PRINT option is specified, the READ attributes are used during PRINT.

PROMPT statement for menus

The PROMPT statement specifies a text string for each menu selection option. You supply the text for each prompt and Model 204 automatically generates the selection number to accompany each prompt.

The menu selection number is displayed at columns 2 and 3. The numbers are assigned in sequence, beginning with 1, and incremented by one for each additional prompt.

Syntax

The syntax for the PROMPT statement is:

```
PROMPT {'text' | promptname}
 [AT [COLUMN] n] [TO [COLUMN] m | [LEN m]
 [DP {k | *}]]
 [DEFAULT 'value'] [[READ] attributes]
 [REREAD attributes]
 [PRINT attributes] [ITEMID n]
```

Where

- *text* is a character string enclosed in single quotes with a maximum length of 76 characters.
- *promptname* is a menu variable to which a value is assigned before the menu is displayed with the READ MENU or PRINT MENU statement.
- Location options (optional) listed in the following table specify the position of a prompt on the screen.

Option	**Specifies…**
AT n	Where the title is to begin on the screen.
TO m	Column where the title is to end.
LEN m	Length of a title beginning at the n specified in the AT option. If a menu variable is used to supply the title, and you do not specify a length, the default length is the remainder of the line.
DP k *or* DP *	Number of decimal places displayed in a title. • DP k displays k decimal places for the title. Any additional decimal places are truncated. • DP * displays all decimal places.

- COLUMN (optional). If an AT option is not specified in a PROMPT statement, the prompt begins at column 5. If an AT option is specified, the prompt begins at the position indicated by n. A prompt cannot start before column 5. If the difference between the AT and TO specifications or the LEN specification is less than the length of the string, the string is truncated.

If the specified length exceeds the length of the string, spaces are used to pad the end of the prompt.

Note: The prompt string is not right-justified at the column indicated by the TO specification.

- DEFAULT lets you provide a default literal value for a named prompt. The value must be enclosed in single quotes. This option eliminates the need to use an assignment statement to set the initial value of a PROMPT variable.
- READ selects the display attributes for the item on execution of a READ statement. The default display attributes for prompts are PROTECTED, DIM, and VISIBLE.

  Note: You incur an error message if you use an UNPROTECTED attribute in a prompt.
- PRINT selects the display attributes for the item on execution of a PRINT statement. If no PRINT option is specified, the READ attributes are used during PRINT.

Usage

Normally, each PROMPT statement starts a new line of text on the screen. However, multiple-line prompts can be handled by repeating *text* or *promptname* for a single PROMPT statement. An example of this is shown below.

```
PROMPT 'EMPLOYEE ADD' AT 10 '(NEW PERSONNEL ONLY)' -
        AT 10
```

This produces the following output:

```
1     EMPLOYEE ADD
      (NEW PERSONNEL ONLY)
```

SKIP statement for menus

The SKIP statement passes over one or more lines on the screen between menu options. Do not to exceed the total number of lines on a screen when using SKIP. Skipping beyond 23 lines is not allowed.

Syntax

The syntax for the SKIP statement is:

```
SKIP n LINE[S]
```

where *n* is a positive integer that specifies the number of lines to be left blank on the screen. For example:

```
SKIP 2 LINES
```

causes two blank lines before the next menu line.

The SKIP %variable LINE(S) option is not supported for this application of the SKIP statement.

MAX PFKEY statement for menus

The MAX PFKEY statement specifies the maximum PF key value associated with a particular menu.

If a MAX PFKEY statement is present and the terminal operator presses a PF key with a value greater than n in response to a READ or PRINT MENU statement, the PF key value is divided by n. If the PF key value is evenly divided by n, n is returned to %menuname:SELECTION. If the PF key value is not evenly divided by n, the value of the remainder is returned to %menuname:SELECTION. For example, if MAX PFKEY 12 is specified, Model 204 returns PF13 through PF24 as PF1 through PF12.

Syntax

The syntax for the MAX PFKEY statement is:

```
MAX PFKEY n
```

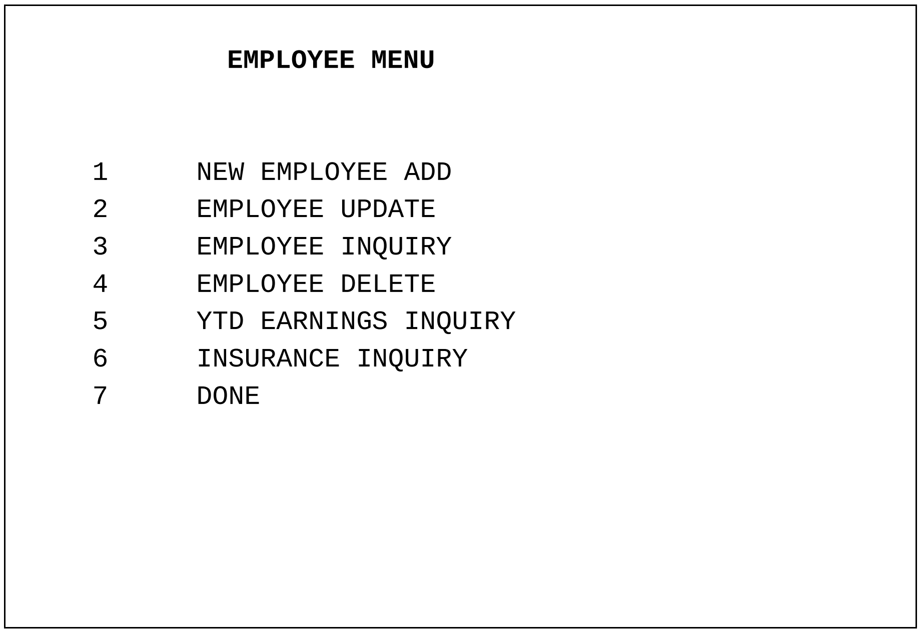
where *n* is a number from 1 to 255. If *n* exceeds 255, a default value of 255 is used. The statement can appear anywhere in the menu definition after the title line. Only one MAX PFKEY statement is allowed for each menu definition.

Menu definition example

In this example, the user defines the format of the EMPLOYEE MENU selection screen:

```
MENU PERSONNEL
TITLE 'EMPLOYEE MENU' AT 13 BRIGHT
MAX PFKEY 12
SKIP 2 LINES
PROMPT 'NEW EMPLOYEE ADD' AT 10
PROMPT 'EMPLOYEE UPDATE' AT 10
PROMPT 'EMPLOYEE INQUIRY' AT 10
PROMPT 'EMPLOYEE DELETE' AT 10
PROMPT 'YTD EARNINGS INQUIRY' AT 10
PROMPT 'INSURANCE INQUIRY' AT 10
PROMPT 'DONE' AT 10
END MENU
```

The preceding statements result in a menu in the format shown in Figure 22-1 when displayed by the request.



```
             EMPLOYEE MENU


     1      NEW EMPLOYEE ADD
     2      EMPLOYEE UPDATE
     3      EMPLOYEE INQUIRY
     4      EMPLOYEE DELETE
     5      YTD EARNINGS INQUIRY
     6      INSURANCE INQUIRY
     7      DONE
```

Figure 22-1. Sample menu created by menu statements

Menu manipulation

Menu manipulation involves reading a previously defined menu and then accepting input items from the terminal operator.

Menu manipulation statements

You can initialize and display a menu and accept a response from the terminal operator. You can specify these statements anywhere in the User Language request, except within a menu or screen definition.

Menus are manipulated using the following statements:

Statement	**Displays…**
MODIFY	Changed attributes of a menu item during request execution.
PREPARE MENU	Reinitialized menu.
PRINT MENU	Menu on a terminal or as USE output.
READ MENU	Menu and accepts a response from the terminal operator.

An example program using these statements is described in “Menu manipulation example” on page 22-28.

READ MENU statement

The READ MENU statement lets you display a defined menu on the screen and accept user selections from that menu. When the menu is displayed, the cursor automatically is positioned under the first menu selection number.

Syntax

The syntax for the READ MENU statement is:

```
READ [MENU] menuname [ALERT] [TITLE
 {'text' | %variable}
 [AT [COLUMN] n] [TO [COLUMN] m | LEN m]
 [attributes]]
```

Where

- *menuname* refers to a menu previously defined by a set of menu definition statements.
- ALERT sounds the audible alarm on the terminal when the menu is displayed. ALERT is ignored and no warning is issued if the terminal is not equipped with an alarm.
- TITLE specifies a new title to override the menu title specified in the TITLE statement of the original menu definition. This new title is effective only for the current READ statement. The title specified in this option can be either a character string (‘text’) or a variable that has been set to a string before the READ MENU statement is executed.

AT, TO, LEN, and attributes options for READ MENU

- The AT, TO, LEN, and attributes options are identical to those described in “Location options” on page 22-16. If these options are omitted, the AT, TO, LEN, and attribute options defined for the title in the menu definition are used.

PRINT MENU statement

The PRINT MENU statement lets you display a menu on a terminal or as USE output with all menu items protected.

When the PRINT MENU statement is evaluated, and output is to an IBM 3270 or compatible terminal, the menu is displayed as it normally would appear during a READ except that the tab keys are ineffective. The terminal operator presses the ENTER key or a PF key to complete the operation of the PRINT. The key that the operator pressed is returned to the request.

Syntax

The syntax for the PRINT MENU statement is:

```
PRINT [MENU] menuname [ALERT] [TITLE
 {'text' | %variable}
 [AT [COLUMN] n] [TO [COLUMN] m | LEN m]
 [attributes]]
```

Where

The ALERT, TITLE, AT, TO, LEN, and attributes options are identical in “Location options” on page 22-16.

MODIFY and PREPARE MENU statements

MODIFY statement

The MODIFY statement changes the display attributes of a menu item during the execution of a User Language request.

A MODIFY statement changes only those attributes that you wish to change (for example, from VISIBLE to INVISIBLE); the statement leaves other attributes unchanged. If ALL is specified or the FOR clause is omitted, the new attributes apply to both READ and PRINT.

Syntax

The syntax for the MODIFY statement is:

```
MODIFY %menuname:itemname

 [TO] attributes [ALL | [FOR] READ | PRINTS]
```

Usage notes

Consider the following when using the MODIFY statement:

- The UNPROTECTED attribute is not allowed for titles and prompts.
- The PREPARE MENU statement restores display attributes to their original state.

PREPARE statement

The PREPARE statement reinitializes a menu. PREPARE can be issued at any point in a request to restore values that were altered by assignment and MODIFY statements. The PREPARE statement restores:

- A cursor that was moved to select a menu option back to its original position
- Specified default values (those indicated by DEFAULT options) to the title and prompts
- Null values to prompts specified with variable prompt names that do not have default values
- Original display attributes if the attributes were overridden by MODIFY statements

Syntax

The syntax for the PREPARE statement is:

```
PREPARE [MENU] menuname
```

Where

menuname refers to a menu previously described by a set of menu definition statements.

Use with global menus

Defining a menu as global affects the order in which you should issue PREPARE statements. See “Clearing the GTBL work area” on page 20-28 for a discussion of performance considerations related to declaring and clearing global objects from GTBL.

Menu manipulation example

In this example, the user defines a procedure that prompts the terminal operator for the next function to be performed, sets a global variable, SELECTION, with the user’s response, and uses the conditional include capability to include the appropriate procedure.

```
PROCEDURE PERSONNEL.APPLICATION
BEGIN
MENU PERSONNEL
TITLE ’EMPLOYEE MENU’ AT 13 BRIGHT
MAX PFKEY 12
SKIP 2 LINES
PROMPT ’NEW EMPLOYEE ADD’ AT 10
PROMPT ’EMPLOYEE UPDATE’ AT 10
PROMPT ’EMPLOYEE INQUIRY’ AT 10
PROMPT ’EMPLOYEE DELETE’ AT 10
PROMPT ’YTD EARNINGS INQUIRY’ AT 10
PROMPT ’INSURANCE INQUIRY’ AT 10
PROMPT ’DONE’ AT 10
END MENU
READ MENU PERSONNEL
IF $SETG (’SELECTION’,%PERSONNEL:SELECTION) THEN
    PRINT ’GLOBAL TABLE FULL’
END IF
END
IF SELECTION=1,ADD.EMPLOYEE
IF SELECTION=2,UPDATE.EMPLOYEE
IF SELECTION=3,INQUIRE.EMPLOYEE
IF SELECTION=4,DELETE.EMPLOYEE
IF SELECTION=5,INQUIRE.EARNINGS
IF SELECTION=6,INQUIRE.INSURANCE
IF SELECTION=7,CLEANUP.EMPLOYEE
END PROCEDURE
```

Defining screens

You define and format screens using screen definition statements. Formatting involves specifying a title and all prompts to be displayed on the screen, and describing input items to be entered by the terminal operator.

You must include a screen definition in the User Language request for every screen to be displayed by that request.

Screen definition format

The statements described on the following pages allow you to logically define a screen. Each screen is defined using the following format:

```
SCREEN screenname
screenline
        .
        .
        .
END SCREEN
```

The logical screen definition must begin with a SCREEN statement and end with an END SCREEN statement. Between these statements, you describe the titles, prompts, input areas, and blank lines that are to appear on the screen. To do this, you specify a series of screenlines that describes the contents of a particular line or portions of lines on the screen.

Screen definition statements

Screenlines are composed of the following statements:

Statement	**Description**
DEFAULT	Specifies the various screen item defaults.
INCLUDE	Includes a stored procedure within a screen definition.
INPUT	Describes an input item entered by the terminal operator.
MAX PFKEY	Specifies the maximum PF key value associated with a menu.
PROMPT	Defines a particular text prompt.
SKIP	Skips one or more lines on the screen.
TITLE	Specifies the screen title.

Screenlines

Each screenline normally corresponds to a single line on the physical panel. The User Language compiler regards a screenline as a logical input line. If the definition of a screenline is longer than a single request input line, use a hyphen for continuation.

Comments or blank lines included in a screen definition (between SCREEN and END SCREEN statements) are ignored by Model 204.

You can include any number of screenlines in a screen definition.

Logical and physical panels

The screen actually displayed to the terminal operator depends on the logical panel and physical panel.

Logical screen

A logical screen (panel) is that part of the logical screen definition which is ended by a NEW PAGE or by an END SCREEN statement. These statements control where the display screen is to end.

Physical screen

A physical screen (panel) is that part of the logical panel which fits on the user’s physical terminal as determined by the MODEL parameter. The full-screen formatting feature defines screens independently of the terminal operator’s 3270 terminal model; the correct length is displayed when the request is evaluated. If the prompts and/or input items specified in a screen definition exceed the number of lines available on a physical panel, Model 204 automatically constructs an overflow panel for the remaining lines. The overflow panel has the same title as the original screen.

SCREEN and END SCREEN statements

SCREEN statement

The SCREEN statement must be the first statement in a screen definition. SCREEN signals Model 204 that statements preceding the END SCREEN statement define a logical screen.

Syntax

The format of the SCREEN statement is:

```
[DECLARE] SCREEN screenname
 [GLOBAL [PERMANENT | TEMPORARY]
 | [PERMANENT | TEMPORARY] GLOBAL | COMMON]
```

Where

- *screenname* is a user-defined string from 1 to 255 characters in length. Every screen name defined in a request must be unique. The screen name specified in the SCREEN statement is assigned to the screen being defined; it subsequently can be referenced in screen manipulation statements.
- *DECLARE* and *COMMON* are used if the screen is a common element in the request. Specifying COMMON (or specifying neither COMMON nor GLOBAL) results in the screen being stored in FSCB and processed as it would have been prior to Version 2.2. Common elements, as well as DECLARE and COMMON, are discussed in detail in “Sharing common elements” on page 12-15.
- *GLOBAL* specifies that the screen is stored in GTBL. Global menus have an implied scope of COMMON (see above).

  Note: Do not specify the GLOBAL attribute for multi-page (logical pages) screens.
- *PERMANENT (PERM)* can only be used with global screens. PERM global screens persist across request boundaries (they are maintained in GTBL even after a request has been terminated).
- *TEMPORARY (TEMP)* can only be used with global screens. TEMP global screens are allocated in GTBL, but are deleted at request termination. An example of using a TEMP global screen would be when you do not need to pass the screen to another request; using TEMP global screens eliminates the need for you to explicitly delete globals that do not need to persist.

END SCREEN statement

Every SCREEN statement must have a corresponding END SCREEN statement, which terminates the definition of a screen. The END SCREEN statement must be the last statement in a screen definition.

TITLE and PROMPT statements for screens

TITLE statement

The TITLE statement specifies a character string to be displayed as the screen title on the first line of the screen.

If you do not specify a TITLE statement

If a TITLE statement is not defined in a menu definition, a menu title can be specified when the menu is displayed (see the section titled “READ SCREEN statement” on page 22-52). If no title is specified, the following system menu title is displayed:

```
INDICATE NUMBER FOR DESIRED SELECTION
```

Syntax If a you specify a TITLE statement, the statement must be the first menuline defined in a menu definition. The format of the TITLE statement is:

```
TITLE {'text' | promptname} [AT [COLUMN] n]
 [TO [COLUMN] m | [LEN m] [DP {k | *}]]
 [DEFAULT 'value']
 [[READ] attributes] [REREAD attributes]
 [PRINT attributes]
```

Where

- *text* is a character string enclosed in single quotes with a maximum length of 79 characters.
- *promptname* is a screen variable to which a value can be assigned before the screen is displayed with the READ SCREEN, REREAD SCREEN, or PRINT SCREEN statement.
- REREAD selects the display attributes for the title on execution of a REREAD statement. If no REREAD option is specified, the READ attributes are used during REREAD.
- The other options for the TITLE statement used with screens are the same as the options described in “TITLE statement for menus” on page 22-16.

PROMPT statement

The PROMPT statement specifies the text to be displayed as a prompt for the terminal operator. A prompt usually asks the operator to enter a particular data value.

Syntax The syntax for the PROMPT statement is:

```
PROMPT {'text' | promptname} [AT [COLUMN] n]
 [AT [COLUMN] m | [LEN m] [DP {k | *}]]
 [DEFAULT 'value']
 [[READ] attributes] [REREAD attributes]
 [PRINT attributes]
 [ITEMID n]
```

Where

- *text* is a character string in single quotes; use up to 78 characters.
- *promptname* is a screen variable to which a value can be assigned before the screen is displayed with the READ SCREEN, REREAD SCREEN, or PRINT SCREEN statement.

  Note: Each prompt can start a new line of text on the screen or multiple prompts can be defined for one screenline. However, each prompt must occupy its own position on a screenline.
- REREAD selects the display attributes for the prompt on execution of a REREAD statement. If no REREAD option is specified, the READ attributes are used during REREAD.
- ITEMID assigns a number from 1 to 32767 to a prompt. This number is used for reference by the cursor variable %screenname:ITEMID. For more information about %screenname:ITEMID, refer to the section titled “Cursor handling” on page 22-56.
- The other options for the PROMPT statement used with screens are the same as the options for the PROMPT statement used with menus, as described in “PROMPT statement for menus” on page 22-18.

INPUT statement

The INPUT statement defines the characteristics of an input value to be entered by the terminal operator, usually in response to a prompt. An INPUT statement typically is defined in conjunction with one or more PROMPT statements. Any mixture of INPUT and PROMPT statements can be included on the same logical screenline in a screen definition.

Syntax The format of the INPUT statement is:

```
INPUT inputname [AT [COLUMN] n]
 [TO [COLUMN] m | [LEN m] [DP {k | *}]]
 [UPCASE | NOCASE] [DEFAULT 'value']
 [DEBLANK | NODEBLANK] [PAD WITH 'c']
 [automatic validation options]
 [[READ attributes] [REREAD attributes]
 [PRINT attributes]
 [TAG [attributes] [WITH 'c']] [ITEMID n]
```

Where *inputname* indicates the name of the data item entered by the terminal operator.

AT, TO, LEN, and DP options

AT, TO, LEN, and DP options are described in the following table:

Option	Specifies…
AT n	Where the title is to begin on the screen.
TO m	Column where the title is to end.
LEN m	Length of a title beginning at the n specified in the AT option. If a menu variable is used to supply the title, and you do not specify a length, the default length is the remainder of the line.
DP k *or* DP *	Number of decimal places displayed in a title. • DP k displays k decimal places for the title. Any additional decimal places are truncated. • DP * displays all decimal places.

COLUMN keyword

COLUMN (optional). If the AT specification is omitted, and the input item is the first item on the screenline, the input area begins in column 2. If it is not the first item on the screenline, the input area begins, following one blank space, to the right of the text displayed by the previous PROMPT statement or space reserved by the previous INPUT statement.

When both AT and TO are specified, space is reserved for the input item on the current line at locations n through m inclusive. For example, the ADDRESS item defined below:

```
INPUT ADDRESS AT 15 TO 29
```

reserves space on the screen in positions 15–29. The length of the ADDRESS is thus 15 characters (m-n+1).

When AT and LEN are both specified, space is reserved for the input item on the current line starting at position n and extending m characters (to position n+m). The following statement also reserves space on the screen in positions 15–29:

```
INPUT ADDRESS AT 15 LEN 15
```

UPCASE and NOCASE options

UPCASE and NOCASE allow one item (or multiple items) on a screen to be specified with a case attribute that differs from the rest of the screen. UPCASE forces case translation and NOCASE prevents case translation. See “Case translation” on page 22-35.

Case translation

Case translation specifies whether screen input is displayed on the screen as entered, or displayed as uppercase. If case translation is in effect, all input to the screen is displayed in upper case, regardless of whether it is entered in upper or lower case. For input items, case translation specifies how the screen input is stored in Model 204.

Case translation is based on:

- The setting for *LOWER
- The setting of UPCASE or NOCASE
- The value of the LANGUSER parameter

The settings of UPCASE and NOCASE take precedence over the setting for *LOWER and *UPPER. Even if *UPPER is in effect, if an item is specified as NOCASE, it is stored or displayed in the case in which it was entered. Conversely, if *LOWER is in effect and an item is specified as UPCASE, the item is translated and stored or displayed in uppercase.

When UPCASE is specified, the value of the LANGUSER parameter is checked to determine the language that is in use, and, consequently, the correct case translation rules to use. See the *Model 204 Language Support Summary* for more information on language processing and the LANGUSER parameter.

DEFAULT option

The DEFAULT option specifies a default or initial value for an input item. The value must be enclosed in single quotes.

This default is shown as the value of the input item when the screen is first displayed. It can be changed at that time by the terminal operator.

If the default string exceeds the length specified for the input item, the string is truncated. If the length of the default string is less than the specified length, the default is padded with spaces at the end. The value must be enclosed in single quotes.

This option eliminates the need to use an assignment statement to set the initial value of an INPUT variable.

DEBLANK or NODEBLANK option

The DEBLANK option specifies that leading and trailing blanks on the input item are to be removed from any value entered into the input item by the terminal operator. The NODEBLANK option specifies that leading and trailing blanks are to be retained. DEBLANK is the system default.

Default values and values assigned to an input item by the request are not affected by this option.

PAD WITH ‘c’ option

The PAD WITH ‘c’ option displays an input item as padded with a user-specified character. The keyword WITH is optional; ‘c’ can be any character you select.

This option helps to delineate the length of an input item. It differs from the DEFAULT option because the pad character are not retained as part of the input (any leading or trailing pad characters are removed).

Be careful when combining the PAD WITH ‘c’ option with the DEBLANK option (which is the default). If both options are present, any leading or trailing mixture of blanks and pad characters are removed. If PAD WITH ‘c’ is specified with NODEBLANK and the input item begins or ends with blanks, any adjacent pad characters are not removed.

Automatic validation options

Automatic validation options specify the criteria by which Model 204 automatically validates an input value before that value is used in the request. See “Automatic validation options for INPUT” on page 22-39 for more information.

READ option

The READ option selects the display attributes for the item on execution of a READ statement. The default display attributes for input items are UNPROTECTED, DIM, and VISIBLE.

REREAD option

The REREAD option selects the display attributes for the item on execution of a REREAD statement. For an input item, the REREAD attributes are used only if the item is not tagged. If no REREAD option is specified, the READ attributes are used during REREAD.

PRINT option

The PRINT option selects the display attributes for the item on execution of a PRINT statement. If no PRINT option is specified, the READ attributes other than UNPROTECTED are used during PRINT.

TAG option

The TAG option specifies the display attributes for an item on the execution of a REREAD statement when the item is tagged. You also can specify the character to be displayed as the error indicator when an input value does not meet the automatic validation criteria. If a TAG option is not specified for a particular input item, the default error indicator, an asterisk (*), is displayed in column 80. If more than one value on the same screenline is in error, the error indicator of the rightmost item is used as the tag character.

Syntax

The format of the TAG option is:

```
TAG [attributes] [WITH ’c’]
```

where:

- *attributes* indicate any valid display attribute supported by the user’s terminal (see the section titled “Display attributes” on page 22-8). The default TAG attributes are UNPROTECTED, BRIGHT, and VISIBLE.
- *WITH ‘c’* specifies the character (c) to be displayed in column 80 as the error indicator; any EBCDIC character can be specified. The default character is * (asterisk).

  The use of BRIGHT, highlighting, and/or color attributes for tagged items is especially useful when more than one input item appears on the same line on the screen. If you do not wish an error indicator to be displayed, a space must be explicitly specified as the tag character, as shown below.

  ```
  INPUT CODE TAG BRIGHT WITH ’ ’
  ```

ITEMID option

The ITEMID option assigns a number from 1 to 32767 to an input item. This number is used for reference by the cursor variable %screenname:ITEMID. For more information on %screenname:ITEMID, as it relates to cursor handling, refer to “Reserved cursor variables” on page 22-56.

Automatic validation options for INPUT

Automatic validation options specify the criteria by which Model 204 automatically validates an input value before that value is used in the request. Automatic validation is performed after the DEBLANK option is processed. For more information on automatic input validation, refer to “Input validation” on page 22-7.

Automatic validation is performed by specifying any of the options listed below in an INPUT statement. If validation requirements are not met, the error is tagged and an error indicator is displayed if the screen is redisplayed with a REREAD statement.

Option	**If specified for an input item…**
ALPHA	Only upper- and lowercase alphabetic characters (for the language specified by the LANGUSER parameter) are accepted.
ALPHANUM	Only upper- and lowercase alphabetic characters (for the language specified by the LANGUSER parameter) and/or digits (0-9) are accepted.
MUST FILL	Either the number of characters entered must match the specified length, or no characters must be entered. Otherwise, an error is signaled. • If both the DEBLANK and MUST FILL options are specified and the terminal operator has entered leading and/or trailing spaces, the input item is tagged as in error. • REQUIRED must be specified if null input values are not acceptable.
NUMERIC	Only digits (0-9), minus sign, and one decimal point are accepted. A minus sign must be in the leftmost character position.
ONEOF value[,value]…	Value entered must be one of a specified set of values. You can include any number of values in the list; values are separated by commas. Enclose values containing commas or spaces with quotation marks. (for example, ‘1,000,000’). The following example specifies that an entered value is one of the New England states: `INPUT STATE ONEOF MA, CT, RI, NH, VT, ME`

Option	If specified for an input item…

RANGE	Only numbers or character strings in the specified range are allowed in the input value. RANGE performs range checking in the form:
	`[NUMERIC] [RANGE lo [TO] hi [AND lo [TO] hi] …`
	If the NUMERIC option is specified immediately before RANGE, the number entered must be greater than or equal to lo, and less than or equal to hi. If the input value falls within either range, it is accepted.
	Numbers used with this option can be integers and/or fractions.
	If the NUMERIC option is not specified, a character range is assumed. The string you specify must be between lo and hi, inclusive, in EBCDIC collating sequence.
	A string that contains more than one word (containing either a space or a character that normally ends a word, such as right parenthesis or =) must be quoted, as shown below.
	`INPUT NAME RANGE ADAMS TO ’LE BLANC’`
	Both types of range option allow any number of ranges to be specified using AND. The following statement tests for two distinct ranges of numbers:
	`INPUT CODE NUMERIC RANGE 1001 TO 2999 AND 5001 TO 6999`
	If the input value falls within either range, it is accepted.

REQUIRED	Item is tagged when a null value is entered.

VERIFY ‘characters’	Only the specified set of characters can be included in the input value. Each character can appear in the input value any number of times and in any order. For example, the following statement ensures that input IDs contain only, but not necessarily all, the specified characters
	`INPUT ID VERIFY ’ABCDXYZ.:-&’`
	Thus, the following IDs are acceptable:
	`A.B`
	`DD`
	`XCZ -:`
	and the following IDs are unacceptable:
	`A,B`
	`MNO`
	`AB*`

Multiple validation criteria

With the exception of REQUIRED, multiple validation criteria for a single input item are treated as if they were connected by OR Boolean operators. If an input value satisfies any one of the item’s criteria, the value is accepted. Note that both numeric and nonnumeric validation criteria can be specified for a single input item. If REQUIRED is specified, the input value is checked for a null value before the input value is validated against other item criteria.

DEFAULT statements

Default statements define various screen item defaults, in some cases replacing the need for specification on each screen item.

The DEFAULT TITLE (or PROMPT or INPUT) statement sets the READ, REREAD, TAG, and PRINT display attributes globally for screen items. The DEFAULT CURSOR statement defines the default cursor position for the defined screen.

Syntax The format for the DEFAULT TITLE (or PROMPT or INPUT) statement is:

```
DEFAULT {TITLE | PROMPT | INPUT [DEBLANK | NODEBLANK]

 [PAD WITH 'c']

 [LEN m [DP [k | *}]]

 [UPCASE | NOCASE]

 | [TAG [attributes] [WITH 'c']]}

 [[READ] attributes] [REREAD attributes]

 [PRINT attributes]
```

Where The options are the same as described on “INPUT statement” on page 22-34.

Scope of DEFAULT TITLE or PROMPT or INPUT statements

The scope of a particular DEFAULT statement is either until another DEFAULT statement for the same screen item type or until the END SCREEN statement. Any display attribute defined explicitly for a screen item overrides the corresponding display attribute specified in the DEFAULT statement.

For example:

```
DEFAULT INPUT READ YELLOW BLINK
     .
     .
     .
INPUT MSGLINE READ WHITE
```

results in WHITE and BLINK attributes for the MSGLINE item.

The DEFAULT statement does not affect items defined earlier in the screen.

Syntax The format for the DEFAULT CURSOR statement is:

```
DEFAULT CURSOR [READ | REREAD | PRINT]

 {ITEMID n | itemname | ROW n COLUMN m}
```

Where *ITEMID*, *itemname*, or *ROW* and *COLUMN* specify where the cursor is positioned initially on the execution of a READ, REREAD, or PRINT SCREEN statement, unless overriding cursor setting information exists (a WITH CURSOR option). For more information on cursor positioning, refer to “Cursor handling” on page 22-56.

The DEFAULT CURSOR statement can appear anywhere in a screen definition; only one DEFAULT CURSOR statement is allowed for each screen definition.

SKIP and NEW PAGE statements

SKIP statement

The SKIP statement passes over one or more lines on the screen.

Syntax

The format for the SKIP statement is:

```
SKIP n LINE[S]
```

where *n* is a positive integer that specifies the number of lines to be left blank on the screen. For example, this statement causes two blank lines to appear before the next screenline:

```
SKIP 2 LINES
```

NEW PAGE statement

The NEW PAGE statement forces a new page for multi-panel screens.

MAX PFKEY statement for screens

The MAX PFKEY statement specifies the maximum PF key value associated with a particular screen.

Syntax

The format for the MAX PFKEY statement is:

```
MAX PFKEY n
```

where *n* is a number from 1 to 255. If *n* exceeds 255, a default value of 255 is used. The MAX PFKEY *n* statement can appear anywhere in the screen definition after the title line. Only one MAX PFKEY statement is allowed per screen definition.

How the pressing of PF keys greater than *n* is handled

If a MAX PFKEY statement is present and the terminal operator presses a PF key with a value greater than *n* in response to a READ, REREAD, or PRINT SCREEN statement, the PF key value is divided by *n*.

- If the PF key value is evenly divided by *n*, *n* is returned to %screenname:PFKEY.
- If the PF key value is not evenly divided by *n*, the value of the remainder is returned to %screenname:PFKEY.

For example, if MAX PFKEY 12 is specified, Model 204 returns PF13 through PF24 as PF1 through PF12.

INCLUDE statement

The INCLUDE statement includes a stored procedure within a screen definition. The stored procedure can contain screen definition statements, followed by any other valid procedure input.

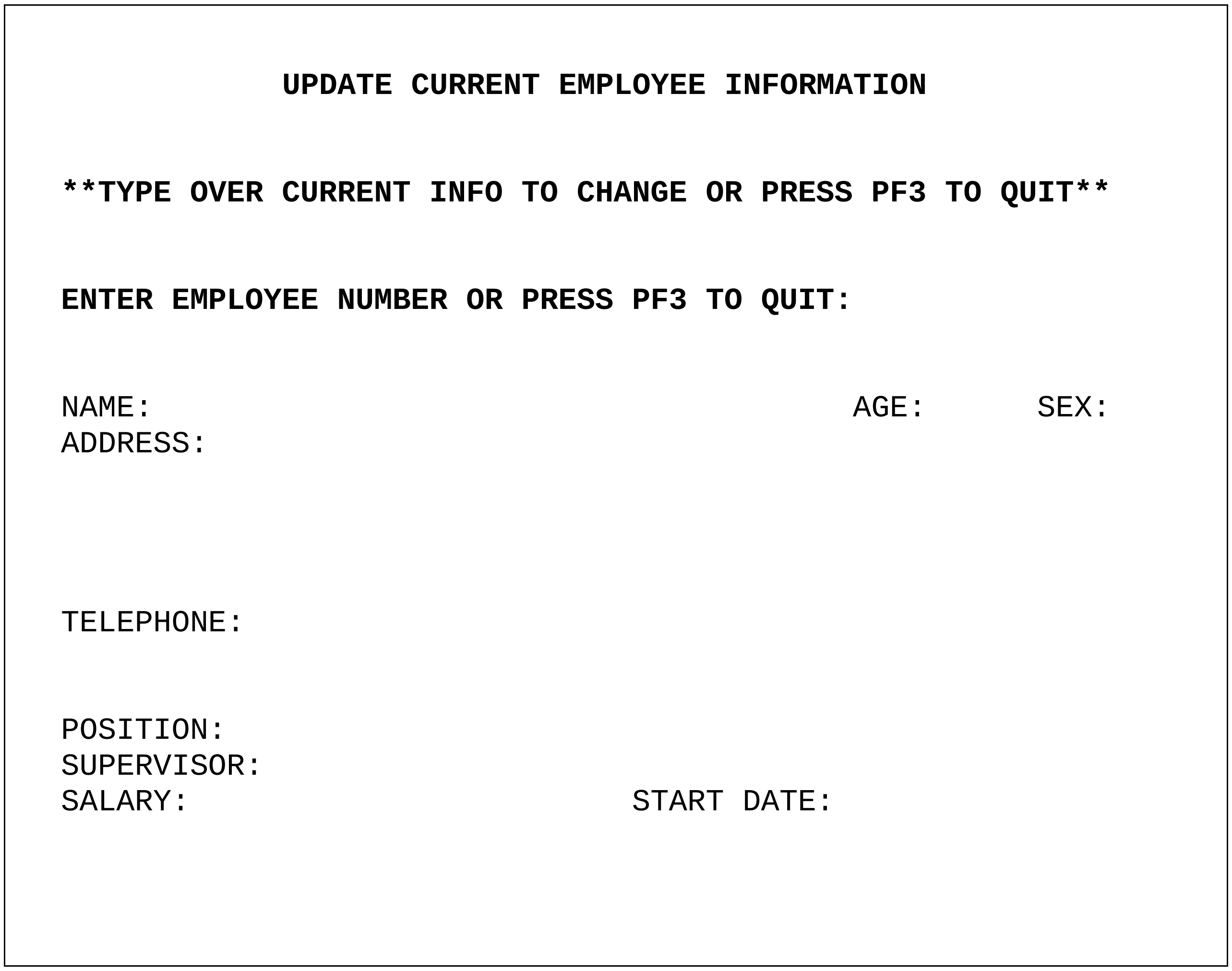
The INCLUDE statement must occur at the beginning of a screen definition line. The format of this statement is the same as that for the INCLUDE statement (see Chapter 13) with the optional IN clause.

Screen definition example

In the following screen definition example, the user defines the format of the UPDATE CURRENT EMPLOYEE INFORMATION screen:

```
SCREEN UPDATE
MAX PFKEY 12
DEFAULT CURSOR READ NBR
TITLE 'UPDATE CURRENT EMPLOYEE INFORMATION' AT 19
BRIGHT
SKIP 2 LINES
*
*THE TEXT FOR MSGLINE IS DEFINED LATER IN THE REQUEST
*
INPUT MSGLINE BRIGHT
SKIP 2 LINES
PROMPT 'ENTER EMPLOYEE NUMBER OR PRESS PF3 TO QUIT:'
-
    BRIGHT INPUT NBR LEN 4 NUMERIC
SKIP 2 LINES
PROMPT 'NAME:' INPUT NAME LEN 50 -
    PROMPT 'AGE:' AT 59 INPUT AGE LEN 2 NUMERIC -
    PROMPT 'SEX:' AT 67 INPUT SEX LEN 1 UPCASE ONEOF
M,F
PROMPT 'ADDRESS:' INPUT ADDRESS1
                  INPUT ADDRESS2 AT 11
                  INPUT ADDRESS3 AT 11
                  INPUT ADDRESS4 AT 11
PROMPT 'TELEPHONE:' INPUT PHONE
SKIP 2 LINES
PROMPT 'SUPERVISOR:' INPUT SUPER
PROMPT 'POSITION:' INPUT POSITION LEN 20 -
PROMPT 'DEPARTMENT NO:' AT 35 INPUT DEPT LEN 5 DP *
PROMPT 'SALARY:' INPUT SALARY LEN 8 DP 2 -
PROMPT 'START DATE:' AT 35 INPUT DATE
END SCREEN
```

The preceding statements result in a screen of the format shown in Figure 22-1 when displayed by a request:



```
              UPDATE CURRENT EMPLOYEE INFORMATION

**TYPE OVER CURRENT INFO TO CHANGE OR PRESS PF3 TO QUIT**


ENTER EMPLOYEE NUMBER OR PRESS PF3 TO QUIT:


NAME:                                        AGE:      SEX:
ADDRESS:




TELEPHONE:


POSITION:
SUPERVISOR:
SALARY:                         START DATE:
```

Figure 22-2. Sample screen created by a SCREEN statement

Screen manipulation

Screen manipulation involves reading a previously defined screen and accepting input items from the terminal operator. You manipulate screens by using screen manipulation statements that:

- Initialize and display a screen
- Accept responses from the terminal operator
- Validate input
- Highlight errors.
- Control cursor placement (refer to “Cursor handling” on page 22-56).

You can specify screen manipulation statements anywhere in the User Language request, except within a screen definition (between a SCREEN and an END SCREEN statements).

Screen manipulation statements

Table 22-2 lists and briefly describes the screen manipulation statements.

Table 22-2. Screen manipulation statements

Statement	Description
CLEAR TAG	Clears the error indicator.
MODIFY	Changes display attributes of a screen item.
PREPARE SCREEN	Reinitializes a screen during request execution.
PRINT SCREEN	Displays a screen on a terminal or as USE output.
READ SCREEN	Displays a screen and accepts responses from the terminal operator.
REREAD SCREEN	Redisplays a screen for correction.
TAG	Highlights invalid responses with an error indicator.

Each of these statements, along with cursor handling, is discussed separately on the pages that follow. In addition, a discussion is provided on the sequence in which screens are evaluated when a READ SCREEN, REREAD SCREEN, or PRINT SCREEN statement is executed. An example illustrating screen manipulation appears at the end of the discussion.

Note: The functions, “$CHKMOD” on page 27-9 and “$CHKTAG” on page 27-14, are particularly useful when performing screen manipulation.

MODIFY and PREPARE statements for screens

MODIFY statement

The MODIFY statement changes the display attributes of a screen item during the execution of a User Language request.

Note these considerations when using the MODIFY statement:

- The UNPROTECTED attribute is not allowed for titles and prompts.
- The PREPARE SCREEN statement restores display attributes to their original state.

Syntax

The format for the MODIFY statement is:

```
MODIFY %screenname:itemname [TO] attributes

 [ALL | READ | [FOR] REREAD | TAG | PRINT]
```

A MODIFY statement changes only those attributes that you wish to change (for example, from BRIGHT to DIM); the statement leaves other attributes unchanged. If ALL is specified or the FOR clause is omitted, the new attributes apply to READ, REREAD, TAG, and PRINT.

PREPARE statement

The PREPARE statement reinitializes the values of all input items and other text displayed on a screen. PREPARE can be issued at any point in a request to restore values that were altered by READ, REREAD, TAG, MODIFY, and assignment statements.

The PREPARE statement performs the following operations:

- Restores specified default values (those indicated by DEFAULT options) to the title, prompts, and input items
- Restores null values to items that do not have default values
- Clears all tags
- Restores the original display attributes if the attributes were overridden by a MODIFY statement

Syntax

The format of the PREPARE statement is:

```
PREPARE [SCREEN] screenname
```

where s*creenname* refers to a screen that was previously described in a screen definition.

Use with global screens

Defining a screen as global affects the order in which you should issue PREPARE statements. See “Performance considerations” on page 20-32 for a discussion of performance considerations related to declaring and clearing global objects from GTBL.

PRINT SCREEN statement

The PRINT SCREEN statement displays a screen on a terminal or as USE output with all screen items protected. With the PRINT statement, a single screen definition can be used for both data entry and reporting purposes.

When the PRINT SCREEN statement is evaluated, and output is to an IBM 3270 or compatible terminal, the screen is displayed on the terminal in panels as normally would appear during a READ. All INPUT items are protected. The terminal operator presses the ENTER key or a PF key to complete the operation of the PRINT. The key that the user presses is returned to the request.

If a USE command has been issued prior to executing a PRINT SCREEN statement, the output is formatted in the same manner as PRINT SCREEN output on a line-at-a-time terminal (item spacing is preserved). For more information, refer to the section titled "Line-at-a-time terminal support" on page 22-63.

Syntax

The format for the PRINT SCREEN statement is:

```
PRINT SCREEN screenname [ALERT] [[WITH] CURSOR]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

where the *WITH CURSOR* attribute option is identical to that described earlier for the READ SCREEN statement, and the other options being the same as the corresponding ones for the READ MENU statement (see "READ MENU statement" on page 22-24).

READ SCREEN statement

The READ SCREEN statement lets you perform these activities:

- Display a user-formatted screen on the terminal.
- Cause Model 204 to wait for the terminal operator to enter appropriate input values. Model 204 then performs automatic validation on those values.
- Make input values available for further validation and processing.
- Optionally redisplay the screen for corrected input.

The first time a particular READ SCREEN statement is executed for a request, the items on the screen are displayed either as null or with the default values specified in the DEFAULT option on the INPUT statement or assigned by the request. Subsequent READ statements for that screen clear all old tags and display the values entered by the terminal operator.

Syntax

The format of the READ SCREEN statement is:

```
READ [SCREEN] screenname [ALERT] [NO REREAD]
 [[WITH] CURSOR]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

Where

- *screenname* refers to a screen previously defined by a set of screen definition statements.
- Except as noted below, the other components of the READ SCREEN statement are the same as the corresponding ones for the “READ MENU statement” on page 22-24.

NO REREAD option

The NO REREAD option cancels the automatic redisplay of the screen when errors are detected in input items. In this case, Model 204 performs the automatic validation and tags invalid items, but the terminal operator is not prompted to correct them.

NO REREAD is used in application requests that specify both automatic validation (using INPUT statement validation options) and manual validation (using other statements in the request). This option allows the user-specified error checking statements within the request to detect errors, tag invalid items, and then redisplay the screen using the REREAD statement. The redisplayed screen shows both the system- and user-tagged errors and allows the terminal operator to correct both types of errors at the same time.

WITH CURSOR option

The WITH CURSOR option positions the cursor using a cursor setting variable. Refer to “Cursor handling” on page 22-56 for a detailed description of how the WITH CURSOR option is used to position the cursor.

REREAD SCREEN statement

The REREAD statement redisplays a screen so that the terminal operator can correct the values on input items tagged as errors.

Note: If no items are tagged, the screen is not redisplayed.

REREAD is the only statement that allows the operator to correct input values tagged as a result of manual validation. The terminal operator can modify items other than tagged items when the screen is redisplayed.

All input is validated again using the automatic validation options on the INPUT statements before it is returned to the User Language request for manual validation. However, the screen is not redisplayed automatically if any of the input fails automatic validation.

Syntax

The format of the REREAD SCREEN statement is:

```
REREAD [SCREEN] screenname [ALERT] [[WITH] CURSOR]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

where *screenname* refers to a screen that was previously described in a screen definition.

The WITH CURSOR attribute option is identical to those described earlier for the READ SCREEN statement; the other options are the same as for corresponding options for the see “READ MENU statement” on page 22-24.

Do not reread a screen which has no TAG fields. You can either explicitly tag a field before using REREAD, or use the following construct:

```
IF $CHKTAG('screenname') THEN
   REREAD SCREEN screenname
   * process screen
ENDIF
```

TAG and CLEAR TAG statements

TAG statement

The TAG statement activates the error indicator associated with an input item.

Syntax The format of the TAG statement is:

```
TAG %screenname:inputname [attributes] [WITH 'c']
```

where *%screenname:inputname* identifies the item to be tagged. The attributes and WITH 'c' options are the same as those described earlier for the TAG option of the INPUT statement.

The last TAG or MODIFY statement executed (referred to below as the "active" statement) defines the attributes.

If the TAG statement is specified…	Then the input item…
Without options	Is tagged with the character specified in the TAG option of the INPUT statement.
Without attributes	Attributes are those specified on the TAG option of the INPUT statement, unless there is an active MODIFY statement that applies to the input item.
With attributes	Attributes are those specified on the TAG statement, even if there is an active MODIFY statement in effect. Furthermore, the NUMERIC and INVISIBLE attributes are preserved, if they are present.

CLEAR TAG statement

The CLEAR TAG statement clears an activated tag for a particular input item, or clears the tags for all input items on a screen.

Syntax The format of the CLEAR TAG statement is:

```
CLEAR TAG {screenname | %screenname:inputname}
```

If you omit *inputname*, all tags for the specified screen are cleared. If you include *inputname*, only the tag associated with the specified input item is cleared.

Cursor handling

Model 204 provides the following cursor handling features that can be used in the manipulation of screens:

Cursor handling feature	During the execution of a READ, REREAD, or PRINT SCREEN statement, selects…
Cursor setting	Initial position of the cursor
Cursor sensing	Final position of the cursor.

Cursor setting or sensing is implemented by using special reserved cursor handling variables.

Reserved cursor variables

The reserved variables listed below are used when performing cursor setting or sensing. The format for each is:

```
%screenname:variable
```

These variables can be used to tie a cursor to a particular screen item or to a specific row and column number.

Variable	Use to perform cursor…
COLUMN	Setting and sensing. The range values for COLUMN is 1 through 80. The column number is relative to the beginning of the screen as it is defined logically, which is the same as its physical display position.
ITEMID	Setting and sensing. The number corresponds to an ITEMID option specified in a prompt or input definition.
ITEMNAME	Setting only. %screenname:ITEMNAME sets the cursor under a particular screen item. The value is a character string identical to one of the item names in the screen definition.
ROW	Setting and sensing. The row number is relative to the beginning of the screen as it is defined logically. Defined rows (TITLE, PROMPT, INPUT, or SKIP statements) in the logical screen are all included in the calculation of %screenname:ROW. A logical row number is the same as its physical display position only within the first display panel of a SCREEN. For example, on a 24-row terminal, the first 23 rows of the SCREEN (including the title) are rows 1 through 23. The title line of the second panel is row 1. The second line of the second panel is row 24. Rows would continue to be numbered in this manner, with each physical panel accounting for 22 rows. The title line is always row 1.

After the execution of a READ, REREAD, or PRINT statement, the cursor variables contain values. These values are not reset by a PREPARE statement.

It is the responsibility of the User Language request to reset the variables, if necessary, before the next READ, REREAD, or PRINT statement.

Using cursor setting

You specify the cursor setting by assigning values to cursor setting variables and using the WITH CURSOR option on the READ, REREAD, or PRINT SCREEN statements.

The order of precedence for where the cursor is placed under:

1. The item named by %screenname:ITEMNAME, if the variable's value is not null and appears in the screen definition.
2. The item identified by %screenname:ITEMID, if the variable's value is not 0 and appears in the screen definition.
3. The position indicated by %screenname:ROW and %screenname:COLUMN, if one or both variables are not zero and both are within the screen definition.

   Note: If one of the variables is 0, a 1 is forced.
4. The position indicated by the DEFAULT CURSOR statement in the screen definition if all cursor setting variables have a null, 0, or invalid value, or if there is no WITH CURSOR option.
5. The first UNPROTECTED input item on the current panel for READ, under the first UNPROTECTED tagged input item on the current panel for REREAD, or in the upper left corner of the screen for PRINT.
6. If the above conditions fail, the cursor is placed in the upper left corner.

The screen panel with the position selected by these rules is the first one displayed, with the cursor properly positioned. Variables that have higher precedence must be cleared explicitly by the request to allow variables that have lower precedence to take effect.

Be careful when coding requests that loop around READ, REREAD, or PRINT SCREEN statements with the WITH CURSOR option. If the variables are not explicitly changed before each statement, the previous input cursor position is used for the new output position.

Example of cursor setting

```
BEGIN
SCREEN A
TITLE ' SCREEN A' AT 2
SKIP 2 LINES
DEFAULT INPUT TAG RED WITH ' '
PROMPT ' ENTER PFKEY3 TO QUIT:' -
       INPUT NMR
PROMPT ' ENTER YOUR FAVORITE COLOR ' -
```

```
       AT 11 INPUT COLOR
PROMPT ’ ENTER YOUR FAVORITE CAR ’ -
       AT 15 INPUT CAR
END SCREEN
%A:ITEMNAME=’CAR’
READ SCREEN A WITH CURSOR
IF %A:PFKEY EQ 3 THEN
   STOP
END IF
REREAD SCREEN A
END
```

Using cursor sensing

After a READ, REREAD, or PRINT SCREEN statement is executed, the cursor sensing variables contain information on the position of the cursor in the logical screen definition. Cursor position variables are set as follows:

Variable	Value
%screenname:COLUMN	Logical screen column
%screenname:ITEMID	Value of ITEMID on the PROMPT or INPUT statement, or 0
%screenname:ITEMNAME	Null
%screenname:ROW	Value of 1 on the TITLE statement, or logical screen row

Model 204 returns ROW and COLUMN values beyond the bounds of the logical screen if that is where the terminal operator has placed the cursor. The request is responsible for validating the input cursor position.

READ, REREAD, and PRINT evaluation sequence

The rules that govern the evaluation of the READ SCREEN, REREAD SCREEN, and PRINT SCREEN statements are:

1. If the cursor has been explicitly set, the logical panels preceding the one holding the cursor position are validated on a READ SCREEN or REREAD SCREEN statement. If a READ SCREEN statement is executed with the automatic reread enabled and tagged items exist, the first tagged logical panel is displayed first (skip to Step 4).
2. If there is no automatic reread to perform, the logical panel that contains the cursor position is handled (skip to Step 4).
3. Otherwise, the first logical panel is handled.
4. If the cursor has been explicitly set, the physical panels preceding the one holding the cursor position is validated on a READ SCREEN or REREAD SCREEN statement.
5. If there is no automatic reread to perform, the physical panel with the cursor position is displayed (skip to Step 7).
6. Otherwise, the first physical panel is displayed.
7. Input is accepted for the current physical panel.
8. The physical panel is automatically validated on a READ SCREEN or REREAD SCREEN statement. If the automatic reread is enabled and any item is tagged, the panel is redisplayed (return to Step 7).
9. If the ENTER key is used, the next physical panel, if any, is displayed for input (return to Step 7).
10. If a PF key is pressed, all remaining physical panels are validated for a READ SCREEN or REREAD SCREEN statement. If there are any tagged items and the automatic reread is in effect, the next physical panel following the panel on which the PF key was pressed is displayed (return to step 7). If an automatic reread is not in effect, each remaining logical panel is validated. If any item fails validation, the first physical panel on that logical panel is displayed (return to Step 7).
11. If ENTER continues to be pressed, all remaining logical panels in the logical screen are handled.
12. %screenname:ROW, %screenname:COLUMN, %screenname:PFKEY, and %screenname:ITEMID all reflect values from the last physical panel displayed on the terminal.

Screen manipulation example

The following request uses the UPDATE CURRENT EMPLOYEE INFORMATION screen defined in “Screen definition example” on page 22-46 to allow the terminal operator to display an employee record. The operator selects the record to be displayed by entering the employee number. Once the record is displayed, the operator can change the employee information.

Once all the necessary information has been changed, the record is updated. A message is immediately displayed in order to notify the operator as to whether the update process was successful.

```
BEGIN
*
* DEFINE SCREEN FOR EMPLOYEE UPDATE PROMPTING FOR
EMPLOYEE
* NUMBER
*
SCREEN UPDATE
MAX PFKEY 12
DEFAULT CURSOR READ NBR
TITLE ’UPDATE CURRENT EMPLOYEE INFORMATION’ AT 19
BRIGHT
SKIP 2 LINES
INPUT MSGLINE BRIGHT
SKIP 2 LINES
PROMPT ’ENTER EMPLOYEE NUMBER OR PRESS PF3 TO QUIT:’
-
    BRIGHT INPUT NBR LEN 4 NUMERIC
SKIP 2 LINES
PROMPT ’NAME:’ INPUT NAME LEN 50 -
    PROMPT ’AGE:’ AT 59 INPUT AGE LEN 2 NUMERIC -
    PROMPT ’SEX:’ AT 67 INPUT SEX LEN 1 UPCASE ONEOF
M,F
PROMPT ’ADDRESS:’ INPUT ADDRESS1
                  INPUT ADDRESS2 AT 11
                  INPUT ADDRESS3 AT 11
                  INPUT ADDRESS4 AT 11
PROMPT ’TELEPHONE:’ INPUT PHONE
SKIP 2 LINES
PROMPT ’POSITION:’ INPUT POSITION
PROMPT ’SUPERVISOR:’ INPUT SUPER
PROMPT ’POSITION:’ INPUT POSITION LEN 20 -
PROMPT ’DEPARTMENT NO:’ AT 35 INPUT DEPT LEN 5 DP *
PROMPT ’SALARY:’ INPUT SALARY LEN 8 DP 2 -
PROMPT ’START DATE:’ AT 35 INPUT DATE
END SCREEN

*
* PROMPT USER FOR EMPLOYEE ID NUMBER. IF NOT FOUND
THEN
* GIVE MESSAGE.
* IF FOUND THEN GET RECORD AND DISPLAY CURRENT INFO.
```

```
*
READ.SCRN:READ SCREEN UPDATE
CANCEL:   IF %UPDATE:PFKEY EQ 3 THEN
                SKIP 3 LINES
                PRINT 'REQUEST CANCELLED - RECORD -
                    NOT UPDATED' AT COLUMN 10
                STOP
            END IF

GET.EMP:    FIND ALL RECORDS FOR WHICH
                ENUM=%UPDATE:NBR
            END FIND
CT.RECS:    COUNT RECORDS IN GET.EMP
            IF COUNT IN CT.RECS EQ 0 THEN
                %UPDATE:MSGLINE =-
                '**THERE IS NO EMPLOYEE WITH THIS -
               RECORD NUMBER PLEASE CHANGE OR PRESS
PF3**'
                READ SCREEN UPDATE
                %UPDATE:MSGLINE=''
                JUMP TO CANCEL
            END IF
*
* RETRIEVE CURRENT INFO TO DISPLAY ON SCREEN
*
GET.INFO: FOR EACH RECORD IN GET.EMP
              %UPDATE:NAME=NAME
              %UPDATE:AGE=AGE
              %UPDATE:SEX=SEX
              %UPDATE:PHONE=PHONE
              %UPDATE:POSITION=POSITION
              %UPDATE:DEPARTMENT NO=DEPT
              %UPDATE:SUPER=SUPERVISOR
              %UPDATE:SALARY=SALARY
              %UPDATE:DATE=SDATE
              FOR %I FROM 1 TO 4
                  %ADDR='UPDATE:ADDRESS' WITH %I
                  :%ADDR=ADDRESS(%I)
              END FOR
          END FOR

          %UPDATE:MSGLINE =-
          ' **TYPE OVER CURRENT INFO TO CHANGE OR -
          PRESS PF3 TO QUIT**'
          READ SCREEN UPDATE
          %UPDATE:MSGLINE=''
          IF %UPDATE:PFKEY EQ 3 THEN
              SKIP 3 LINES
             PRINT 'REQUEST CANCELLED - RECORD NOT -
                  UPDATED' AT COLUMN 10
              STOP
          END IF
          IF NOT $CHKMOD('UPDATE') THEN
              SKIP 3 LINES
             PRINT 'NO CHANGES WERE ENTERED - RECORD
-
                   NOT UPDATED' AT COLUMN 10
              STOP
          END IF

*
* UPDATE CURRENT EMPLOYEE RECORD
*
UPDT.REC: FOR EACH RECORD IN GET.EMP
              CHANGE NAME TO %UPDATE:NAME
              CHANGE AGE TO %UPDATE:AGE
              CHANGE SEX TO %UPDATE:SEX
              CHANGE PHONE TO %UPDATE:PHONE
              CHANGE POSITION TO %UPDATE:POSITION
              CHANGE DEPARTMENT NO TO %UPDATE:DEPT
              CHANGE SUPERVISOR TO %UPDATE:SUPER
              CHANGE SALARY TO %UPDATE:SALARY
              CHANGE DATE TO %UPDATE:SDATE
              FOR %I FROM 1 TO 4
                  %ADDR='UPDATE:ADDRESS' WITH %I
                  CHANGE ADDRESS(%I) TO :%ADDR
              END FOR
          END FOR
*
* DISPLAY SUCCESSFUL COMPLETION MESSAGE
*
UPDT.GOOD: SKIP 3 LINES
           PRINT 'EMPLOYEE RECORD SUCCESSFULLY
UPDATED' -
               AT 10
           STOP
END
```

Line-at-a-time terminal support

Although the full-screen features described in this chapter are oriented to the use of IBM 3270 video display terminals, requests that invoke menus and screens can be run from line-at-a-time terminals as well. Model 204 automatically changes its mode of full-screen operation to match the characteristics of the line-at-a-time terminal.

The following discussion applies to all line-at-a-time terminals.

Menus

Menu output is handled on a line-by-line basis. When a READ MENU statement is issued, each line of the menu is displayed on a separate terminal line, preceded by the system-generated selection number. At the end of the menu, Model 204 prompts the terminal operator by displaying:

```
ENTER SELECTION, CR FOR n
```

where *n* is the current value of %menuname:SELECTION. If the terminal operator enters an invalid menu selection number, Model 204 reprompts.

If the terminal operator decides not to view the entire menu, the attention key (for example, BREAK or ATTN) can be pressed to stop the display. Model 204 then skips automatically to the ENTER SELECTION prompt shown above.

Pressing the attention key at any time other than when a menu is being displayed either causes the ON ATTENTION unit (if present) to be executed or the request to be cancelled.

When a PRINT MENU statement is issued and output is sent to a line-at-a-time terminal, each line of the menu is sent a row at a time. Item spacing from the menu definition is preserved.

Screens

All screen inputs and outputs are handled on a line-by-line basis. When a READ SCREEN statement is issued, each TITLE, PROMPT, and INPUT statement included in the screen definition is handled as a separate interaction with the terminal. The titles, prompts, and default input values are shown in the order of appearance in the screen definition. Titles and prompts are displayed on separate lines on the terminal. Consecutive prompts that have no intervening INPUT statements are displayed on the same line. Every INPUT statement included in the request causes the terminal operator to be prompted for input.

If an input item already has a value when the READ is issued, the value is displayed as follows:

```
DEFAULT:value
```

The user can indicate acceptance of the default value by pressing carriage return. Alternatively, the user can enter a new value.

If an invisible item is specified, the item is preceded by a user-specified prompt, or by the default prompt:

```
INVISIBLE INPUT:
```

Default values of invisible items are not displayed.

If an input value has automatic validation options specified for it; see the section titled “Screen manipulation” on page 22-48. Model 204 checks the response immediately, and the user is not able to proceed until a correct value is entered. Note, however, that if the NO REREAD option was specified in the READ statement, Model 204 moves on to the next item whether or not the terminal operator enters the correct value.

The operation of the REREAD SCREEN statement is similar to that of READ SCREEN. REREAD displays the title:

```
THE FOLLOWING ITEMS HAVE INVALID VALUES:
```

before the user-specified title. Following these titles, only items that have been tagged as errors are displayed. The prompt preceding the tagged input item in the screen definition is displayed, followed by the tagged input.

When a PRINT SCREEN statement is issued and output is to a line-at-a-time terminal, each line of the screen is sent a row at a time. Item spacing from the screen definition is preserved.

23

Application Subsystem Development

In this chapter

Overview

Although only a system manager can define a subsystem, the determination of a subsystem's options and components typically also involves the file manager and application developer. This chapter focuses on subsystem facility topics most relevant to the application developer:

The Subsystem Management facility of Dictionary lets you define a collection of procedures to Model 204 as a subsystem and to assign certain characteristics to that subsystem.

Advantages of subsystems

The following table summarizes the advantages of subsystems over other User Language procedure applications:

Advantage	**Subsystems…**
Minimal end-user intervention	Require minimal knowledge of Model 204. The end user need not know what files and procedures exist for the application. The subsystem is invoked simply by entering the subsystem name as a Model 204 command.
Driver facility	Eliminate the need for user-written drivers containing conditional INCLUDEs based on a global variable. This driver facility leads to smaller, more modular procedures that are easier to maintain and enhance.
Performance improvements	Improve performance by saving and reloading compiled User Language requests, called precompiled procedures. Depending upon how often precompiled procedures are included, 20–90% of the operating costs of a Model 204 application can be saved.
Error handling facilities	Trap and handle Model 204 errors in a single, centralized routine. Each subsystem can have one error procedure that is invoked each time a Model 204 error occurs during that subsystem's processing. Model 204 provides facilities for determining the type of error that caused the error procedure to be invoked.
Security facilities	Either allow or restrict access to the subsystem.
Parallel Query Option/204 compatibility	Can be defined to allow referral to remote files and scattered groups.

Subsystem definition

The characteristics and components of a subsystem are defined to Model 204 by the system manager during a process called ***subsystem definition***. The defined options and components are stored in the system file CCASYS. Once a subsystem has been defined, all Dictionary users can display the options and components through Dictionary. For more information about:

- Displaying a subsystem definition, see the *Model 204 System Manager's Guide*.
- Defining subsystems that refer to remote files and scattered groups, see the *Parallel Query Option/204 User's Guide*.

Subsystem design components

During subsystem definition, the components listed below can be defined. These components impact various aspects of subsystem design. The following table summarizes the required component designations.

Component	Subsystem design requires designation of…
Command line global variable	(Optional) parameter global variable. The parameter global variable allows any parameters specified by a user during a subsystem login to be stored in this variable and retained when control is transferred to another subsystem.
Communication global variable and exit value	Communication global variable and exit value. The communication global variable is used to transfer control from one procedure to another. The exit value is used to leave the subsystem. Optionally, a reserved global variable is available for transferring control between subsystems.
Error global variable	Error global variable. If an error occurs while the subsystem is executing, a three-character error code is stored in this variable. This code can then be used by an error procedure to determine the action to be taken by the subsystem.
Prefix designations	Two prefixes for procedure names. These prefixes allow Model 204 to determine whether a procedure can be saved in its compiled form for later evaluation.
Processing components	Specific procedures for types of special processing. These procedures allow Model 204 to determine the flow of control within a subsystem.

Command line global variable

A command line global variable allows you to store any parameters specified by an end user during a subsystem login and retain this information when control is transferred to another subsystem. The designation of a command line global variable is optional.

Using the command line global variable

The command line global variable is used in the following manner:

- A user logs into a subsystem by entering the subsystem name followed by the parameter information. The total length of the parameter information entered by the user can consist of as many as 255 characters. (The portion of the command line reserved for parameter information is discarded if no parameter information is defined.)
- The portion of the input following the subsystem name is placed into a command line global variable, which then is available to the application program. For example:

  ```
  PAYROLL parameter1 parameter2
  ```

  is the command that logs the current user into the subsystem named PAYROLL. The string *parameter1 parameter2* is the subsystem command line and is made available to the application via a global variable. If CMDL is the name assigned to the command line global variable, the following statements:

  ```
  BEGIN
    %CMD.LINE = $GETG('CMDL')
    %FIRST.PARM =
       $SUBSTR(%CMD.LINE,1,$INDEX(%CMD.LINE,' ')-1)
  ```

  would assign the contents of the command line used to enter the subsystem to %CMD.LINE and the first parameter of the line to %FIRST.PARM.

Transferring control to another subsystem

The contents of command line global variables are not deleted when control is transferred from one subsystem to another. A request can set the contents of the command line global variable of the destination subsystem before transferring control to that subsystem. The effect is the same as if the parameters were entered on the user’s terminal.

Impact of the UTABLE command

The contents of the command line global variable are not deleted by UTABLE commands which normally delete the contents of GTBL, as long as the UTABLE command is issued from within a subsystem.

Communication global variable

Transferring control

A communication global variable lets you transfer control at two levels:

From…	Control is transferred by a…
One procedure to another	User-designated global variable.
One subsystem to another	Reserved global variable named XFER.

Transferring control between procedures

Subsystem procedures pass control from one to another through the use of the communication global variable. The communication global variable name for a subsystem is specified in the subsystem definition. Each subsystem procedure must set the value of the communication global variable to the name of the next procedure to be executed.

Example For example, the subsystem AUTOS has procedures PRE-MAIN.MENU and PRE-RPT.PGM and the communication global variable NEXT. PRE-MAIN.MENU is currently executing and wants to pass control to PRE-RPT.PGM. Before PRE-MAIN.MENU finishes, the function $SETG is used to store procedure name PRE-RPT.PGM in NEXT:

```
IF $SETG('NEXT','PRE-RPT.PGM') THEN …
```

After PRE-MAIN.MENU ends, Model 204 examines NEXT and begins executing PRE-RPT.PGM.

Subsystem exit value

A subsystem is exited by setting the value of the communication global variable to the subsystem exit value. The exit value for the communication global variable is also specified in the subsystem definition.

For the AUTOS subsystem used in the preceding example, the exit value of the communication global variable is defined as EXIT. To disconnect the user from the AUTOS subsystem after procedure PRE-RPT.PGM is finished executing, PRE-RPT.PGM assigns the exit value, EXIT, to the communication global variable, NEXT.

```
IF $SETG('NEXT','EXIT') THEN
      .
      .
      .
```

Transferring control between subsystems

One subsystem can invoke another subsystem by transferring control from itself to another subsystem. To accomplish this, you must perform these steps:

1. Set the communication global variable to the value XFER.
2. Set the global variable XFER to the name of the subsystem to which control is being passed.

When the current procedure finishes executing, Model 204 disconnects the user from the old subsystem, transfers control to the new subsystem, and invokes the login procedure for the new subsystem.

Design considerations

You should consider the following factors when coding logic for transferring control between two subsystems:

- The transfer always invokes the login procedure of the subsystem receiving control. For more information on the login procedure, refer to the discussion “Subsystem processing flow” on page 23-27.
- The destination subsystem must be active. The $SUBSYS function should be used to determine if the subsystem to which control is being transferred is active; refer to the discussion on “Subsystem procedure control functions” on page 23-38. If the destination subsystem does not use the automatic start option, (see “Operating options” on page 23-23), the subsystem must be started before control is passed.
- The destination subsystem should not reset LGTBL if any parameters are passed in global variables.
- To return to the original subsystem, a global variable must be set to the name of the original subsystem. The communication global variable and the XFER global variable can then be used with the global variable that stores the subsystem name to return control to the original subsystem.
- The destination subsystem can return control to the procedure that the user was in when the user transferred. To do this, the subsystem must save the name of the procedure in a global variable. In addition, the login procedure must contain logic to return control to the procedure that the user was in.
- If the transferring subsystem is in test mode (see “Security options” on page 23-20), the transferring subsystem stops after it passes control. The destination subsystem is not placed in test mode.

Example

The following request in procedure PRE-SUB.MENU provides an example of subsystem transfer code. CREDIT and AUTOS are two defined subsystems with the automatic start option. (See “Automatic start” on page 23-23.) Subsystem CREDIT transfers control to AUTOS by setting NEXT (the communication global variable) to XFER and the global variable XFER to

AUTOS. After the PRE-SUB.MENU procedure ends, the user is connected to AUTOS.

```
BEGIN
.
.
.
*SELECTION = 4 INDICATES CHOICE OF AUTOS SUBSYSTEM
*
SEL.AUTOS: IF %CREDITMENU:SELECTION = 4 AND -
              $SUBSYS('AUTOS') =1 THEN
                  IF $SETG('XFER','AUTOS') OR -
                      $SETG('NEXT','XFER') OR -
                      $SETG('SUBFROM','CREDIT') OR -
                   $SETG('PROCFROM','PRE-SUB.MENU')
THEN
                      PRINT 'GLOBAL TABLE FULL'
                  END IF
                  JUMP TO GET.NEXT
                    .
                    .
                    .
```

Coding considerations

- Procedures to which control is passed via the communication global variable must be stored in a designated procedure file. The procedure file is the default file for a subsystem application unless the default file is explicitly changed by a DEFAULT command or overridden by an IN clause.
- Each procedure must set the communication global variable to indicate the next procedure to be included. If this variable is not set, an error or loop occurs.
- To exit the subsystem, the communication global variable must be set to the exit value. Server table sizes and other parameters should be reset to the values existing prior to entering the subsystem so that the user is returned to his/her normal operating environment. Parameter values are restored automatically when the automatic login option is used.

Error global variable

The name of the error global variable must be specified in the subsystem definition. Whenever an error is detected that is not trapped by an ON unit, Model 204 automatically sets the subsystem's error global variable to a value which indicates the type of error that occurred.

Error code values

Table 23-1 lists the error global variable values and their corresponding causes. Correct the cause of the error and/or change your error procedure as discussed in "Error procedures" on page 23-10.

Table 23-1. Error global variable values and reasons

Error code	Reason
ATN	User pressed attention
BUG	Evaluation errors
CAN	Request cancellation following BUMP or inactive thread timeout
CNT	Counting errors
FIL - BROKEN	Referenced file is not initialized, full, or physically inconsistent. Check the audit trail to determine the condition of the file.
FIL - NOT OPEN	Referenced file not open
GRP - FTBL	FTBL too small
GRP - NOT OPEN	Group not open
GRP - TEMP FIELD	TEMP group field has wrong type (see "Restrictions for temporary and ad hoc groups in precompiled procedures" on page 23-12)
GRP - TEMP MISMATCH	TEMP group has wrong type
INCLUDE MAX	Maximum iterations value has been exceeded.
HNG	Phone hung up, connection lost
HRD	Hard restart
REC	Record locking table filled up during the load of a precompiled request
SFT	Soft restart
TBL - FSCB	FSCB too small
TBL - NTBL	NTBL too small
TBL - QTBL	QTBL too small

Table 23-1. Error global variable values and reasons (continued)

Error code	Reason
TBL - STBL	STBL too small
TBL - VTBL	VTBL too small

Error procedures

An error procedure must test for different error conditions. The resulting value stored in the error global variable helps the application programmer determine the type of error that occurred.

- For all error codes except ATN, the error procedure should avoid re-executing the procedure that caused the error; otherwise, the error recurs.
- In most cases, the error procedure should display an informational message and set the global communication variable to the exit value.
- If a HNG error code is indicated, all terminal I/O (such as PRINT, READ) is ignored.
- If a HNG, HRD, or SFT error code is indicated, no terminal I/O (such as PRINT or READ) should be attempted. Instead, send a message to the audit trail in an AUDIT statement that indicates the error code encountered.

Considerations for the communications global variable

The communications global variable is ignored and disconnect processing completed when one of the following conditions occurs:

- Error is a soft restart, a hard restart, or a phone hang-up condition.

  If you attempt to set the communications global variable to the name of another procedure, the procedure is not executed.
- No error procedure is specified in the subsystem definition.

An example illustrating how a subsystem error procedure can test for different error conditions is provided in “Error procedure” on page 23-18.

Precompiled and non-precompiled procedures

You can use two types of procedures in a subsystem:

- **Precompiled procedure**—The first time a precompilable procedure is invoked after a subsystem starts, it is compiled and stored for re-use. Because the compiler phase is bypassed each subsequent time the procedure is invoked (except as noted below), precompilation saves both CPU and elapsed time. Exceptions include the following:
  - The procedure is recompiled the first time it is invoked for each SCLASS. The new compilation is evaluated, then discarded. It does not replace the original stored compilation.
  - Further recompilations might be required due to temporary group differences. See “Recompiling precompiled procedures” on page 23-13 for more on this.
- **Non-precompiled procedure**—A non-precompiled procedure is compiled each time it is invoked.

All procedures, whether precompiled or not, are invoked using the communication global variable.

Defining prefixes

During subsystem definition, two prefixes must be defined. The first prefix identifies precompiled procedures; the second identifies non-precompiled procedures. All procedures that are included for the subsystem through the use of the communication global variable (such as the login and main processing procedures) must have names that begin with one of these prefixes.

Contents of subsystem procedures

Subsystem procedures can contain Model 204 commands, a request, multiple requests, continued requests, sections of User Language code (for example, subroutines), or any combination thereof. However, the form of the procedure affects whether the procedure can be precompiled and should be taken into account when the subsystem is designed. Restrictions for precompiled procedures are discussed in detail below.

Shared versions of precompiled procedures

Server I/O can be reduced by allowing users executing shared precompiled procedures to use a shared version of QTBL. See the *Model 204 System Manager’s Guide* for more information.

Restrictions for precompiled procedures

Model 204 must ensure that all of the code compiled and saved for a request with the precompiled prefix is consistent for all loading users. To achieve consistency, Model 204 restricts the way in which certain features are used.

Note the following restrictions for precompiled procedures when designing subsystem procedures:

- Procedures must contain exactly one request. Procedures must not contain any commands other than BEGIN. The last statement must be END or END MORE.
- Requests cannot start with MORE.
- Requests must not refer to files or permanent groups that are not mentioned in the subsystem definition.
- Precompiled procedures can contain the User Language INCLUDE statement. The included procedures must be from a subsystem file. Any code inserted as a result of an INCLUDE statement is subject to all the restrictions for precompiled procedures.
- Compiler table sizes must be the same each time a precompiled procedure is invoked. The UTABLE command should be used carefully.
- Dummy strings (??, ?$, ?&) in precompiled procedures are resolved only during compilation for the first user.
- If a subsystem file is referenced by a precompiled procedure, no user can RESTORE or INITIALIZE the file, or RENAME, DELETE, or REDEFINE a field while the subsystem is active.
- Procedures in UNLOCKed members of a PROCFILE GROUP are *not* precompiled.

Restrictions for temporary and ad hoc groups in precompiled procedures

Precompiled requests can refer to temporary or ad hoc groups as long as the files making up the group are specified in the subsystem definition. A temporary group of the same name must have the same composition characteristics for all loading users, as described below. If this condition is not met, Model 204 sets the error global variable to GRP-TEMP MISMATCH (see “Error global variable” on page 23-9).

- The temporary group for all loading users must be the same type. Model 204 assigns a type, based on the following file conditions:
  - Some files have record security.
  - All files are sorted or all are hashed.
  - The sort or hash key has the same name in all files.

- Fields of the same name in the temporary group must be of the same type. Each field referenced in a temporary group each time a precompiled procedure is invoked must be found in a file. If this condition is not met, Model 204 sets the error global variable to GRP-TEMP FIELD (see “Error global variable” on page 23-9).

  Field definition attributes can change for fields in temporary groups between compile and loading time. The following changes are allowed:

  – If the field is NON-CODED in any file at compile time, it can be CODED in all files at loading time.

  – If the field is BINARY or FLOAT in any file at compile time, it can be STRING in all files at loading time.

  – If the field is non-NUMERIC RANGE in any file at compile time, the field can be NUMERIC RANGE in all of the files at loading time.

Recompiling precompiled procedures

When designing applications which use precompiled procedures and temporary groups, be aware that temporary groups can cause Model 204 to recompile precompiled procedures under certain conditions.

Precompiled procedures are recompiled when the request references a temporary group and the:

- Compiling user’s temporary group consists of files which are smaller than one or more of the files in the loading user’s temporary group of the same name.

- Compiling user’s temporary group has fewer files than the loading user’s temporary group of the same name.

- Update and retrieve privileges do not match those of the compiling user’s temporary group (of the same name).

If one user’s temporary group contains one large file, and another user’s temporary group contains a number of smaller files, it is possible that a precompiled procedure is recompiled every time it is invoked. To prevent constant recompiling when files are of different sizes, compile temporary groups originally with the largest files and the greatest number files you expect to be included in the temporary group.

If, despite precautions, Model 204 must discard and recompile a precompiled procedure, the loading user must have exclusive access to that procedure—no other user can be executing the procedure within the same subsystem. If another user is executing the procedure, the loading user recompiles a private copy of the procedure. Model 204 discards the private copy when execution has completed.

Procedure compilation and Parallel Query Option/204

When a non-precompiled procedure that references remote files is invoked, one or more remote nodes participate in the compilation and evaluation. When a precompiled procedure is invoked, Model 204 loads the procedure on each of the nodes that participated in the original compilation.

When a subsystem member becomes unavailable during evaluation, the appropriate ON unit is activated.

Errors which occur while loading a remote procedure produce error messages which have the prefix RMT in the global error variable.

Saving compilations

As part of the compilation process, a list of remote nodes referenced in the request is generated with the compiled code. When compilation is complete, the compilation is saved along with the list of nodes. Each remote node referenced in the request is sent a signal to save the compilation.

If for any reason a compilation cannot be saved by a server node, the entire save operation fails.

Loading saved compilations

At the client node, the saved remote node reference list is checked to see which nodes are loading the request. When the request is loaded on the client, a signal is transmitted to each referenced server node to load the compilation.

New and missing nodes

A temporary group can be changed so that a node is ***new*** (not previously referenced) or ***missing*** (referenced but no longer available). Table 23-2 shows how new and missing nodes affect recompilations and saves.

Table 23-2. Temporary groups with new and missing nodes

	If there are missing nodes...	**If there are no missing nodes....**
And there are new nodes...	Recompile the procedure, do not save	Recompile and save again
And there are no new nodes...	Just load and evaluate	Just load and evaluate

Recompiling saved requests

Saved requests are always recompiled if a new node is introduced into a temporary group. Recompilation can cause a noticeable delay in response time.

In addition, the following changes in the composition of a subgroup also force recompilation of a request. A ***subgroup*** is the group of files at a server node referenced as a part of a group.

Model 204 recompiles saved requests when:

- The number of files in the subgroup has increased (for example, if a user's request includes a file that was unavailable to the previous user)
- The maximum number of segments in a subgroup has increased

Subsystem procedures

Types of subsystem procedures

Subsystem development involves writing the collection of procedures that make up a subsystem. Subsystem procedures can be categorized as one of four types:

Procedure category	Performs…
Initialization	Specified operations each time the subsystem is initialized.
Login	As the entry point for each user of the subsystem.
Main processing	Specific tasks of the subsystem.
Error	Error handling.

Guidelines and restrictions

- Procedures should be small and modular.
- Included procedures normally are included by using the INCLUDE command. Included procedures cannot be precompiled.
- Non-subsystem files can be opened and referenced only by non precompiled procedures.
- If a subsystem file is referenced by a precompiled procedure, no user can RESTORE or INITIALIZE the file, or RENAME, DELETE, or REDEFINE a field while the subsystem is active.
- A subsystem procedure cannot issue the CREATE command for a subsystem file.
- LXTBL and LFTBL cannot be reset from within a subsystem procedure.
- All DO YOU REALLY WANT TO messages are suppressed and the default action is assumed. The default action for each type of message is listed in the *Model 204 Messages Manual*.

  If you do not wish the default action to be executed, statements to handle a situation that would invoke the message should be added to the procedure.

Initialization procedure

The initialization procedure stores instructions for tasks you need to perform each time the subsystem is initialized. An example of such a task is the initialization of a particular work file.

The initialization procedure is optional. If a subsystem uses an initialization procedure, the procedure name must be specified in the subsystem definition.

Login procedure

The login procedure performs the start up for each user of an application. Every time a user invokes the subsystem, Model 204 automatically includes the subsystem login procedure.

The login procedure name must be specified in the subsystem definition.

Typically, the login procedure is used to store current server table sizes in the global variable table for later reference, issue UTABLE commands to set compiler table sizes for the subsystem, and set the communication global variable to the name of the procedure that displays an initial menu.

Example Here is a sample login procedure:

```
CLEARG
BEGIN
IF $SETG('NTBL',$VIEW('LNTBL')) OR -
   $SETG('VTBL',$VIEW('LVTBL')) OR -
   $SETG('QTBL',$VIEW('LQTBL')) OR -
   $SETG('STBL',$VIEW('LSTBL')) OR -
   $SETG('FSCB',$VIEW('LFSCB')) OR -
   $SETG('LECHO',$VIEW('LECHO')) OR -
   $SETG('NEXT','PRE-MAIN.MENU') THEN
         PRINT 'GTBL FULL'
END IF
END
UTABLE LNTBL=450,LQTBL=2300,LVTBL=600,LSTBL=3300
UTABLE LFSCB=5000
RESET LECHO 0
```

Main processing procedures

Main processing procedures perform the specific tasks of the subsystem. There can be as few or as many main processing procedures as necessary for the subsystem to perform its tasks. Main processing procedures are not specified in the subsystem definition. However, each procedure must follow the procedure naming conventions and subsystem coding rules discussed in this chapter.

Example Here is a sample procedure:

```
BEGIN
MENU MAINMENU
    TITLE 'AUTO INSURANCE SYSTEM (MAIN MENU)' AT 10
BRIGHT
    MAX PFKEY 12
    SKIP 5 LINES
    PROMPT ’MAINTENANCE’ AT 10 BRIGHT
    SKIP 2 LINES
    PROMPT ’REPORTING’ AT 10 BRIGHT
    SKIP 2 LINES
    PROMPT ’EXIT’ AT 10 BRIGHT
END MENU
    %NEXT = ’X’
    REPEAT WHILE %NEXT = ’X’
        READ MAINMENU
        IF %MAINMENU:SELECTION = ’1’ THEN
            %NEXT = ’PRE-MAINT.PGM’
        ELSEIF %MAINMENU:SELECTION = ’2’ THEN
            %NEXT = ’PRE-RPT.PGM’
        ELSE %NEXT = ’NON-FINISH’
        END IF
    END REPEAT
CHK.GTAB: IF $SETG(’NEXT’,%NEXT)THEN
             AUDIT ’GLOBAL TABLE FULL - "NEXT"’
          END IF
END
```

Error procedure

An error procedure, which is optional, performs error handling. This procedure is invoked when a condition occurs that cannot be trapped by the executing procedure (for example, a compiler error or an attention with no ON ATTENTION unit coded). An error procedure tests for different error conditions and determines the next procedure to execute, based on the error code value stored in the error global variable. Terminal I/O in a subsystem error procedure following a BUMP or inactive thread timeout results in cancellation of the procedure.

Example

The following error procedure assumes that the error global variable name is ERRCLASS, the communication global variable name is NEXT, and the exit value of the communication global variable is EXIT.

```
PROCEDURE SYS-ERROR
BEGIN
*
* GET THE VALUE OF THE ERROR CLASS GLOBAL VARIABLE
*
GET.VALUE: %ERRORCODE = $GETG(’ERRCLASS’)
*
* IF THE USER HIT ATTENTION, INCLUDE THE MAIN MENU
SCREEN
*
IF.ERROR: IF %ERRORCODE = ’ATN’ THEN
             %NEXT = ’SYS-MAIN-MENU’
*
* IF PHONE WAS HUNG UP, OR ANY KIND OF RESTART, THEN
AUDIT
* MESSAGE AND EXIT.  DO NOT ATTEMPT ANY TERMINAL I/O
* SINCE USER IS NO LONGER CONNECTED.
*
              ELSEIF %ERRORCODE = 'HNG' OR -
                     %ERRORCODE = 'SFT' OR -
                     %ERRORCODE = 'HRD' THEN
                    AUDIT 'SUBSYSTEM ERROR CODE: ' -
                        WITH %ERRORCODE
                     %NEXT = 'EXIT'
*
* CHECK FOR BROKEN FILE
*
             ELSEIF %ERRORCODE = 'FIL - BROKEN' THEN
                   PRINT ' SUBSYSTEM FILE IS BROKEN'
                  PRINT ' CONTACT YOUR FILE MANAGER
'
                     %NEXT = 'EXIT'
*
* SOME UNACCOUNTABLE ERROR HAS OCCURRED, SET EXIT
* ROUTINE, AND EXIT WITHOUT DOING ANY TERMINAL I/O.
*
              ELSE
                  AUDIT 'NOT ACCOUNTED FOR SUBSYSTEM
ERROR CODE: ' -
                 WITH %ERRORCODE
                 %NEXT = 'EXIT'
          END IF IF.ERROR
*
* SET COMMUNICATIONS VARIABLE TO EXIT VALUE
*
COMM.VAR: IF $SETG('NEXT',%NEXT) THEN PRINT 'GTBL
FULL'
          END IF
END.REQUEST:
END
```

Security options

During subsystem definition, various options are specified for a subsystem. Security options determine subsystem command and file and group privileges assigned to a user.

You can also specify system operation options during subsystem definition. System operation options are discussed in the section titled “Operating options” on page 23-23. For a detailed discussion of subsystem definition options, refer to the *Model 204 System Manager’s Guide*.

Status of subsystem

The status of the subsystem affects the type of subsystem security that is implemented. The subsystem can have one of three status settings:

Status setting	**Allows access to…**
Public	All users. All users who enter a public subsystem are assigned to the single subsystem user class and have the same set of privileges.
Semipublic	All users but permits different privileges to be assigned for each user. In a semipublic subsystem, one subsystem user class can be defined as the default class for all users not specifically assigned to another subsystem user class.
Private	Only to specified users. Using a private subsystem prevents any unauthorized entry into the subsystem. All users who are allowed access must be assigned to one of the defined user classes. Unlike a semipublic subsystem, a private subsystem has no default user class.

User class

The set of privileges assigned to a user is based on the user’s subsystem user class or SCLASS. The SCLASS is used by Model 204 to determine the privileges assigned for each file and group in the subsystem when files are opened for the user. File and group privileges must be specified in the subsystem definition because Model 204 bypasses OPENCTL parameter settings and file passwords when opening subsystem files and groups for each user. Whenever the user invokes the subsystem, he/she is assigned the file and group privileges of that subsystem user class.

The SCLASS also determines whether or not the user can issue any of the subsystem control commands listed below. If Model 204 discovers that the user does not have the correct privileges to issue a command, an error message is displayed.

Subsystem control commands	Directs Model 204 to…
DEBUG SUBSYSTEM	Establish a test environment for a multiuser version of the TEST command extension. The subsystem does not have to be stopped to issue the DEBUG command. When a user enters a subsystem in TEST or DEBUG mode, the user's MSGCTL parameter setting is not changed. All error and informational messages that are not suppressed by the user's MSGCTL setting are displayed on the user's terminal.
START SUBSYSTEM	Activate the subsystem and make it available for use. If the subsystem is inactive when the START command is issued, Model 204 opens all the subsystem files and includes the subsystem initialization procedure.
STOP SUBSYSTEM	Stop the subsystem and make it unavailable for use. Once a subsystem is stopped and all users have exited, then all locking and storage resources held by the subsystem are released and all the subsystem files and groups are closed.
TEST	Establish a single user test environment. The TEST command is extended to TEST DEBUG SUBSYSTEM. The subsystem must be stopped to enter TEST mode.

Note: Several aspects of START SUBSYSTEM and STOP SUBSYSTEM processing are unique to distributed applications. For a discussion, see the *Parallel Query Option/204 User's Guide*.

Processing of security violations

The application subsystem traps security violations that occur while a user is running in a subsystem. File read and update security violations, procedure security violations, and field level security violations are interpreted as compilation or evaluation errors in the error global variable. The audit trail messages produced when the error occurred can be examined in order to identify a compilation or evaluation error as a security violation.

Compiling procedures with a different SCLASS

In the following situation, Model 204 saves User 1's compilation:

1. There are multiple SCLASSes for a subsystem.
2. User 1 saves a procedure under SCLASS 1.
3. User 2, under SCLASS 2, recompiles the procedure.

4. User 2’s global variables contain different information from User 1’s so User 2 tries to open different files or groups than User 1.

However, User 2’s compilation is not saved: User 2 receives an error message:

```
M204.0468: COMPILATION NOT SAVED - reason
```

Operating options

Operating options affect certain aspects of the overall behavior of a subsystem. Operating options are distinct from security options, which are discussed on “Security options” on page 23-20. Subsystem options are also discussed in the *Model 204 System Manager’s Guide.*

The following table lists the operating options and what they determine.

Operating option	**Determines whether…**
Automatic start	Subsystem automatically starts for the first user entering the subsystem.
Locking files and groups for subsystem use	Users from outside the subsystem can open and update subsystem files while the subsystem is active.
Automatic login	Users are automatically logged into Model 204 upon entering a subsystem.
Automatic logout	Users are automatically logged out of Model 204 upon exiting a subsystem.
Automatic COMMIT	Any outstanding updates are committed automatically whenever a subsystem procedure ends and transfers control using the communications global variable.
Message displays	Disconnect, informational, and error messages are displayed for subsystem users.
File usage	Subsystem can run when one or more files used by the subsystem are unavailable for use.

Automatic start

If the automatic start option is selected, Model 204 invokes subsystem initialization when the first user attempts to enter the subsystem. The subsystem can be used without a privileged user first issuing the START SUBSYSTEM command.

If the automatic start option is not selected, subsystem initialization occurs only when the START SUBSYSTEM command is issued. The subsystem is then available for use.

Locking files and groups for subsystem use

If the locking files and groups option is selected, Model 204 prevents users that are not running in the subsystem from opening any of the subsystem files or groups while the subsystem is active.

If the locking files and groups option is not selected, users from outside the subsystem can retrieve, modify, or delete records in a subsystem file while the subsystem is active.

Automatic login

If the automatic login option is selected, Model 204 logs on the user when the user enters a subsystem. The user is logged in using the subsystem name as the Model 204 user ID. If the user already has logged into Model 204 before entering the subsystem, Model 204 first closes all the user's files and logs out the user.

The LOGOUT operations that occur as a result of Automatic Login (both at subsystem Login and Disconnect) ignore the SYSOPT=8 (DISCONNECT on LOGOUT) option.

If the login option is not selected, the user's Model 204 user ID is used during subsystem processing.

Automatic logout

If the automatic logout option is selected, the user is logged out of Model 204 upon exiting the subsystem. This option is particularly useful when combined with the automatic disconnect feature of Model 204.

The START SUBSYSTEM command ignores the SYSOPT=8 option for the Automatic Logout subsystem.

If the automatic logout option is not selected, Model 204 logs the user out of the subsystem in one of three ways:

- If the user was previously logged into Model 204, Model 204 restores the user's original user ID and returns the user to the Model 204 command environment.
- If the user was not previously logged into Model 204, the user is logged out of Model 204.
- The SYSOPT=8 option causes the user to be disconnected from Model 204.

Automatic COMMIT

If the automatic COMMIT option is selected, Model 204 automatically issues a COMMIT statement for any outstanding updates whenever a subsystem procedure terminates and transfers control using the communication global variable. If the COMMIT option is not selected, the application must issue the COMMIT statement to commit any pending updates.

Message displays

Model 204 provides these message display options for subsystem users:

- Disconnect message display
- Model 204 informational message display
- Model 204 error message display

When a message display option is selected, messages of that type are displayed on the user's terminal. If a message display option is not selected, all messages of that type are not displayed on the user's terminal. Note that if the display of any Model 204 type message is suppressed, messages for the corresponding type are not displayed on the user's terminal, but are written to the audit trail file (CCAAUDIT).

Typically, subsystem applications are written so that all messages displayed on the user's terminal are produced by the subsystem and Model 204 messages are suppressed.

File usage

Mandatory vs. optional members

Files and permanent groups contained within the subsystem definition can be designated as ***mandatory*** (the default) or ***optional*** members of the subsystem:

- **Mandatory members**—Mandatory files or groups are automatically opened by Model 204 when a user logs into a subsystem and automatically closed when the user leaves the subsystem. Subsystem requests can assume that all mandatory files are open and that they are physically consistent. The user's file privileges are those defined in the subsystem definition for the current user's SCLASS. The opening of a mandatory member cannot be prevented by the subsystem administrator with the STOP FILE command when the subsystem is active. A mandatory member cannot be accessed by another copy of Model 204 until the entire subsystem is stopped.

  If a subsystem procedure issues an OPEN or CLOSE command for a mandatory member, the command is ignored by Model 204 and the user's current privileges are not changed.

- **Optional members**—Optional files or groups provide the ability for a file or group to be stopped by a subsystem administrator (using the STOP FILE command) without stopping the entire subsystem. If a member is defined as optional, it is not automatically opened during the subsystem login. It must be opened by the application by using an OPEN/OPENC statement or command. The file privileges assigned are those specified in the subsystem definition for the current user's SCLASS. The member is closed (for that user only) when the user leaves the subsystem if a CLOSE command has not been issued.

When an optional member is not in use, it can be processed by another copy of Model 204.

Requests that reference mandatory or optional members can be precompiled. Files not contained in the subsystem definition can be opened and referenced within a subsystem application, but the requests that reference those files cannot be precompiled.

Automatic vs. manual members

Subsystem files and permanent group members can also be designated automatic or manual:

- An ***automatic member*** is a subsystem group or file that is opened automatically when the subsystem is started or when a user enters the subsystem.
- A ***manual member*** is a group or file that must be opened explicitly by the OPEN or OPENC command.

Mandatory files cannot be designated manual. Optional files can be designated either automatic or manual.

Permanent vs. temporary groups in subsystem definitions

The GROUP parameter of the subsystem definition applies ONLY to permanent groups. Temporary group names cannot be used. To include temporary group members in a subsystem definition, and thus to enable their use in precompiled code, the members of the temporary group should be individually specified in the subsystem definition.

Summary of file definition options

Table 23-3 summarizes the subsystem file definition options.

Table 23-3. Subsystem file definition options

Subsystem definition option	Automatic open and close?	Pre-compiled code?	Start/stop file command allowed?	File privileges assigned
Mandatory	Must be automatic	Yes	No	SCLASS
Optional	Can be automatic or manual	Yes	Yes	SCLASS
None	Cannot be automatic	No	Yes	File Password

Subsystem processing flow

To design a subsystem, you must be familiar with the flow of control that occurs during subsystem processing. Subsystem processing typically involves the following phases:

Type of processing	During this processing…
Initialization	Subsystem is started and an optional initialization procedure is included.
Login	User is logged into the subsystem; the user's privileges are determined by the subsystem definition. The appropriate required files and groups are opened for access.
Driver	Procedures that make up the main body of the application are included.
Disconnect	All files and groups are closed for the user, who is then logged out of the subsystem.
Error	An optional error procedure is included. The type of error that occurred is available to the error procedure. For many types of errors, the error procedure can resume normal driver processing.

Initialization processing

Initialization processing is invoked when the subsystem is started. A subsystem is started by the START SUBSYSTEM command, or, if the start option is indicated in the subsystem definition, when the first user enters the subsystem.

During subsystem initialization, Model 204 finds the subsystem definition and opens only required subsystem files and groups. If a required file or group cannot be opened, the subsystem initialization procedure terminates and the user is returned to command level.

One of the subsystem components opened during initialization is the procedure file (or group, if a multiple-procedure group has been specified). The procedure file or group must contain all of the subsystem procedures that are included by the subsystem through the communication global variable. Model 204 scans the subsystem procedure file or group for all procedures whose names begin with either of the subsystem procedure prefixes.

The subsystem initialization procedure is included at this time. This is the only time during subsystem processing that the initialization procedure is executed.

If no error occurs, Model 204 adds the subsystem name to the list of active subsystems. At this point, the subsystem is initialized and ready for use.

Login processing

Login processing is invoked when a user enters a subsystem. If the automatic login option is indicated in the subsystem definition, Model 204 logs on the user using the subsystem name as the user ID. If the automatic login option is not indicated, the user's Model 204 user ID remains in use.

Model 204 next finds the user's subsystem user class definition in CCASYS and opens only the required subsystem files and groups with the privileges that are found for that user class. The MSGCTL parameter automatically is set for the user according to the subsystem definition.

Model 204 sets the communication global variable to the name of subsystem login procedure and proceeds into driver processing.

Driver processing

Model 204 determines which procedure to include next by examining the value of the communication global variable. The procedure name must be one of the names located by the scan of the procedure file during subsystem initialization.

If either the global variable or the procedure name cannot be found, the subsystem's error procedure is included. If an error procedure is not specified in the subsystem definition, the user is disconnected from the subsystem.

If the procedure name is found, Model 204 determines which prefix begins the procedure name. Processing then occurs as follows:

- If the procedure name begins with the non-precompiled prefix, Model 204 includes the procedure for compilation and evaluation.
- If the procedure name begins with the precompiled prefix, Model 204 verifies whether the procedure was compiled previously with the set of privileges defined by the user's subsystem user class.

  Once the compilation status of the procedure is determined, processing is as follows:

  – If the procedure was not previously compiled successfully for the set of privileges defined by the user's subsystem user class, Model 204 includes the procedure for compilation and evaluation. If compilation is successful and no previous compilation was saved for the procedure, the contents of the compiler tables are saved in the system file CCATEMP.

  – If the procedure was previously compiled successfully for the user's privilege set, Model 204 loads the contents of the compiler tables from CCATEMP and evaluates the request.

Model 204 repeats driver processing until the value of the communication global variable is set to the exit value specified in the subsystem definition. When the communication global variable is set to the exit value, Model 204 proceeds into disconnect processing.

Disconnect processing

Disconnect processing is invoked when the subsystem application sets the communication variable to the exit value, when an error occurs with no subsystem error procedure, or when a subsystem user is restarted by Model 204. During disconnect processing, Model 204 closes all required subsystem files and groups for the user, as well as any optional files and groups that have not been closed by the application.

Depending upon whether the automatic logout option is indicated, the user is then either logged out of Model 204 or returned to the Model 204 command environment.

Error processing

Error processing is invoked whenever a Model 204 error occurs that cannot be handled by the procedure being executed at the time. When an error is detected, Model 204 sets the value of the error global variable.

If the subsystem has a defined error procedure, the error procedure is included at this time. If the subsystem does not have a defined error procedure, Model 204 proceeds into disconnect processing.

If the error trapped by the subsystem is a soft restart, a hard restart, or a terminal disconnect condition, the error procedure is invoked. The communication global variable is ignored when the error procedure completes and Model 204 proceeds with subsystem disconnect processing.

Parallel Query Option/204 considerations

This section introduces several terms and concepts which are unique to subsystems that reference remote files and scattered groups. These concepts, and related design considerations, are discussed in greater detail in the *Parallel Query Option/204 User's Guide*.

Remote file access

Parallel Query Option/204 provides access to remote files under the Subsystem Management facility by allowing the system manager to define client and service subsystems:

- A ***client subsystem*** is the subsystem a user is running in when requesting access to remote data.
- A ***service subsystem*** is the subsystem on a server node that a client user's service thread is logged into.

A service subsystem definition is stored in the CCASYS file on each node that the client subsystem accesses. The name of a subsystem must be the same at each node. The location of the client node is included in the subsystem name to uniquely identify it to the server node.

Node availability

A server node can be ***available*** or ***unavailable*** to a client subsystem.

- A node is available if the service subsystem has been successfully started.
- If the service subsystem has not been started, it does not have a subsystem definition structure accessible to the client and is therefore unavailable.

A node can only be marked unavailable during start processing if there are mandatory members on a server node and the service subsystem cannot be started. If this happens, start processing also fails on the client node.

Client subsystems attempting to access service subsystems that are not started receive an error message from the server node.

A previously available node can become unavailable when:

- Resumption of communication fails after recovering from a system failure.
- A user attempts to log into the service subsystem by logging into the client subsystem, the service subsystem definition is not found, and at least one mandatory member resides on that node.
- A user attempts to open a file on a node where the user was not previously logged in.

The user is automatically logged into all associated service subsystems when entering a subsystem that contains remote files. If the service subsystem is unavailable on a node, the user cannot be logged in.

File and group availability

The ***members*** of a subsystem are files and permanent groups. With Parallel Query Option/204, members can be either automatic or manual:

- An ***automatic member*** is a subsystem group or file that is opened automatically when the subsystem is started or when a user enters the subsystem.
- A ***manual member*** is a group or file that must be opened explicitly by the OPEN or OPENC command.

Members can also be either mandatory or optional:

- A ***mandatory member*** must be open in order to access a subsystem. Mandatory members cannot be manual.
- An ***optional member*** is not required for subsystem access (start and login processing can succeed without it).

At any given time a member can be ***open*** or ***closed*** to a subsystem or to a user within a subsystem. The following sections explain the conditions under which the different kinds of members are accessible to APSY subsystems and their users.

Member availability to subsystems

Automatic members of subsystems are always opened by the START SUBSYSTEM command or by SUBSYSTEM LOGIN. At the end of START processing, each automatic member is open unless either the START or OPEN failed.

Manual members of subsystems are in the closed state at the completion of START SUBSYSTEM processing and must be explicitly opened by the user. Manual members become open to the subsystem if an OPEN operation succeeds. If OPEN fails due to node unavailability or for user-specific reasons (for example, if the user’s line goes down) the member remains closed to the subsystem.

If a node becomes unavailable to a subsystem, all automatic subsystem members and all open manual subsystem members residing on the unavailable node are marked disabled.

If a STOP FILE/GROUP command is issued for a manual member on the client subsystem’s node, the member is closed to the client subsystem when the last user closes it. If the member is located on the service subsystem node, the file is closed to the service subsystem when the STOP is complete or the last user closes the file.

Member availability to subsystem users

When a user enters a subsystem, automatic subsystem members are opened.

If a user LOGIN or OPEN operation fails for an optional member, the member is left closed for the user but remains available to the subsystem. If a mandatory member cannot be opened, the user is denied access to the subsystem.

If a user LOGIN or OPEN operation fails for an already open member, the member is left disabled for the user but remains open to the subsystem.

If an automatic mandatory member is closed to the subsystem, new users are not allowed to enter the subsystem.

Manual members of subsystems are closed for a user within a subsystem until the user issues an OPEN command or statement. In this case it does not matter whether the member is open or closed to the subsystem.

If compilation and/or loading of a request fails due to a communications failure, previously opened members on the failing node become disabled to the user.

A user can close optional members at any time by issuing the CLOSE command.

Enabling disabled subsystem files

In the event that a subsystem file or group is marked disabled, you can enable it (after correcting the problem) without having to bring the subsystem down. To do this, use the ENABLE SUBSYSTEM command:

Syntax

```
ENABLE SUBSYSTEM subsysname
 [FILE name AT location | GROUP name]
```

Where

- *subsysname* is the name of the client subsystem
- *location* is the name of the remote node where the file is stored. Note that the location must be explicitly specified; you cannot reference local files with the ENABLE SUBSYSTEM command.

Intentionally disabling a subsystem file

You can make a subsystem file or group (or an entire subsystem, if a file is mandatory) temporarily inaccessible without having to bring the subsystem down, using the DISABLE SUBSYSTEM command:

```
DISABLE SUBSYSTEM sybsysname
 [FILE name AT location | GROUP name]
```

When a file or group is intentionally disabled with the DISABLE SUBSYSTEM command, subsystem behavior is exactly the same as when a communications failure causes the disabling. This behavior is described on “Member availability to subsystem users” on page 23-32.

Trust

As an alternative to the privilege settings normally available through the Subsystem Management facility, the system manager at a service node can control client subsystem access by creating a ***trust*** definition for the client subsystem. If a client subsystem is fully or partially trusted, the trust definition is sufficient for maintaining the relationship with the client; the system manager at the service node does not have to create and maintain a separate set of file and SCLASS definitions for the client subsystem.

For example, suppose a subsystem located on a node named DETROIT (the client node) includes in its definition files located on a node named CLEVELAND (the service node). Further, suppose the subsystem is fully or partially trusted by CLEVELAND. In this case, the file and SCLASS definitions are maintained only on the client node (DETROIT), and the service node (CLEVELAND) needs only to maintain the trust definition.

The four levels of trust available with Parallel Query Option/204 are:

- **Full trust**—Only the subsystem name and location appear on the service node’s definition, which you create on the Subsystem Trust screen.
- **Partial trust**—Along with the subsystem name and location, you can specify maximum file privileges. In this case, the client subsystem is trusted, but the maximum file privileges and field level security levels specified on the Subsystem Trust screen cannot be exceeded.
  - If a user requests file privileges that would exceed the maximum, the service node does not open the file to that user.
  - If a user requests a field level security status that would exceed the maximum, Model 204 automatically resets the request to the allowed level (that is, the maximum) and opens the file to that user.
- **Restricted trust**—For a subsystem that has a restricted trust definition, you make *no entries* on the Subsystem Trust screen. A restricted trust definition is based solely on entries you make on these five screens:
  - Subsystem Activity
  - Subsystem File Use
  - Operational Parameters
  - Subsystem Classes
  - Subsystem Class Users

  The accessibility of service node files to a client subsystem is determined by the SCLASS, user, and file privileges that you specify on these screens.
- **No trust**—no subsystem service definition exists for the subsystem. The client subsystem cannot access any files on the service node *as subsystem*

files. The files on the service node can, however, be accessed from within a client subsystem as individual, non-subsystem files if the following criteria are met:

- Parallel Query Option/204 is installed at both sites.
- Horizon is installed at both sites, and there are link, processgroup, and process definitions connecting the client node to the service node.
- For any given file, the value of the OPENCTL parameter allows remote access (X'02', X'04', or X'08'). See the *Model 204 Command Reference Manual* for detailed information on the OPENCTL parameter.

See the *Parallel Query Option/204 User's Guide* for detailed information on creating and managing trust definitions.

Subsystem design considerations

This section presents coding considerations for subsystem procedures. Some of the guidelines listed below also appear in earlier sections. They are consolidated here for the convenience of the application developer.

Coding considerations

- Procedures should be small and modular.
- Procedures to which control is passed via the communication global variable must be stored in a designated procedure file. The procedure file is the default file for a subsystem application unless the default file is explicitly changed by a DEFAULT command or overridden by an IN clause.
- Each procedure must set the communication global variable to indicate the next procedure to be included. If this variable is not set, an error or loop occurs.
- The communication global variable must be set to the exit value in order to exit the subsystem. Server table sizes and other parameters should be reset to the values existing prior to entering the subsystem so that the user is returned to his/her normal operating environment. Parameter values are restored automatically when the automatic login option is used.
- Included procedures normally are included by using the INCLUDE command. Included procedures cannot be precompiled.
- Non-subsystem files can be opened and referenced only by non-precompiled procedures.
- Precompiled procedures cannot reference PERM groups that are not members of the same subsystem.
- Compiler table sizes must be the same each time a precompiled procedure is invoked. The UTABLE command should be used carefully.
- The contents of the command line global variable are not deleted by UTABLE commands which normally delete the contents of GTBL, as long as the UTABLE command is issued from within a subsystem.
- Distinct group numbers are assigned to optional groups at START SUBSYSTEM time. Those numbers cannot be used by non-subsystem members opened within the subsystem. Thus the NGROUP limit used for earlier releases might be exceeded during either START or OPEN processing inside the subsystem.
- To prevent you from having the wrong file or group privileges in a subsystem, Model 204 closes optional files and groups before entering a subsystem. In earlier releases, optional files and groups were only closed when you left the subsystem.

Users should be aware of the following conditions when coding applications to run under the Subsystem Management facility:

- Dummy strings (??, ?$, ?&) in precompiled procedures are resolved only during compilation for the first user.
- If a subsystem file is referenced by a precompiled procedure, no user can RESTORE or INITIALIZE the file, or RENAME, DELETE, or REDEFINE a field while the subsystem is active.
- A subsystem procedure cannot issue the CREATE command for a subsystem file.
- LXTBL and LFTBL cannot be reset from within a subsystem procedure.
- All DO YOU REALLY WANT TO messages are suppressed and the default action is assumed.

  The default action for each type of message is listed in the *Model 204 Messages Manual*.

  If you do not wish the default action to be executed, you need to add a message handler routine to the procedure containing the statement that invokes the message.

Record locking considerations

Depending upon the subsystem definition, Model 204 might place a share lock on one or more subsystem procedure names or group names.

If the subsystem is defined with permanent groups, Model 204 locks the group names to ensure that the group definitions do not change while the subsystem is running. A share lock is maintained for each group while the subsystem is active.

If subsystem files are defined as unlocked

If the subsystem definition specifies that subsystem files are unlocked, Model 204 locks in share mode each of the subsystem procedures to ensure that the procedures do not change or move. This prevents any user from:

- Issuing the DELETE PROCEDURE command
- Issuing the RENAME PROCEDURE command
- Updating the procedure while the subsystem is active.

Subsystem procedure control functions

The User Language functions, “$SCLASS” on page 27-96 and “$SUBSYS” on page 27-104 can be useful in determining subsystem program control.

$SCLASS function

The $SCLASS function returns the SCLASS name of the current user. $SCLASS is useful when the transfer of control is dependent upon a user’s privileges. For example:

```
BRANCH:  JUMP TO (ADD.REC, VIEW.REC, UPD.REC) -
             %MAIN.MENU:SELECTION
                  .
                  .
                  .
UPD.REC: IF $SCLASS = ’READ’ THEN
             IF $SETG(’NEXT’,’PRE-RPT.PGM’) THEN
                 PRINT ’GLOBAL TABLE FULL’
             END IF
         ELSEIF $SCLASS = ’UPDATE’ THEN
             IF $SETG(’NEXT’,’PRE-MAINT.PGM’) THEN
                 PRINT ’GLOBAL TABLE FULL’
             END IF
                  .
                  .
                  .
```

$SUBSYS function

The $SUBSYS function returns a numeric value indicating the status of a subsystem, or the name of the current subsystem (if no argument is specified). $SUBSYS often is used to determine whether a subsystem is active before a transfer is attempted.

Subsystem development tools

This section describes three Model 204 features that are useful in developing, testing and debugging subsystem procedures:

- The DEBUG and TEST commands, which assist subsystem debugging by allowing you to display the global communications variable and specify the next procedure to be INCLUDEd.
- Multiple procedure file groups, which allow users to change procedures without stopping the subsystem or interfering with other users.
- The Cross-Reference facility, which produces reports on the variable names, global dummy strings, and other language elements used in a specified set of User Language procedures.

Debugging and testing facilities

The Model 204 DEBUG and TEST SUBSYSTEM commands assist you in debugging subsystem code while it is being developed. Both commands display the value of the communication global variable, prompt you for changes, and display since-last statistics.

DEBUG differs from TEST SUBSYSTEM in the following ways:

- DEBUG can be executed by more than one user in the same subsystem at the same time; TEST SUBSYSTEM can only be executed by a single user.
- To execute TEST SUBSYSTEM, the user must stop the subsystem, which will be restarted in single user mode as a result of issuing the TEST command. To execute DEBUG, the user does not have to stop the subsystem. The DEBUG command can be issued for any subsystem that has been started, or for any subsystem that has the AUTOSTART feature.
- Since-last statistics are provided automatically with DEBUG; they are optional with TEST.

In general, the DEBUG command is more convenient for subsystem developers in a multiuser environment.

To execute the DEBUG command, the user must be named to an SCLASS which has been granted either the TEST or DEBUG privilege. The DEBUG privilege does not entitle the user to execute the TEST command.

Syntax

The format of the DEBUG command is:

DEBUG **SUBSYS**TEM *subsystemname* [*parameters*]

The extended format of the TEST command is:

TEST [DEBUG] [STATS] [SUBSYSTEM] *subsystemname parameters*

Where

- *DEBUG* specifies that the communication global variable is displayed on the user's terminal before the next procedure is included. The user then has the option of specifying a different procedure to be included next. If the user presses ENTER without specifying a different procedure, the procedure whose name is currently displayed is included next.
- *STATS* specifies that since-last statistics are displayed on the user's terminal after each procedure is evaluated. Since-last statistics are described in the *Model 204 System Manager's Guide*.
- *SUBSYSTEM* specifies that the word following the keyword is the name of a subsystem and that parameters follow the subsystem name. This keyword can be used to eliminate confusion when DEBUG or STATS is the name of a subsystem or subsystem parameter.
- *parameters* specifies the parameters to be stored in the command line global variable. The parameter information can be as many as 255 characters in length.

For more information on the DEBUG and TEST commands, refer to the *Model 204 Command Reference Manual*.

Multiple procedure files

If "PROCFILE = *" is specified when a group is created, then several files in a group can contain procedures. When the INCLUDE command is executed in the context of a multiple procedure file group, files are searched in a fixed order determined by the original CREATE GROUP command.

Multiple procedure file groups make it possible to change subsystem procedures without having to stop the subsystem. This is accomplished by setting GROUP=Y for the subsystem PROCFILE and by specifying NUMLK=*n* (where *n* is less than the number of files in the group). The application subsystem only locks procedures in the last *n* files in the search order determined by CREATE GROUP.

The multiple procedure file option also allows different users to make changes to procedures in the same subsystem without interfering with each other. The PROCFILE GROUP specified in the subsystem definition must correspond to a PERM GROUP. However, any individual user can create or open a TEMP GROUP with the same name as the subsystem's PROCFILE GROUP. If a user has such a TEMP GROUP open and enters a subsystem, then the application subsystem uses the TEMP GROUP instead of the subsystem's PERM GROUP.

The following restrictions apply to the use of TEMP GROUPs to store application subsystem procedure files:

- The user's SCLASS must have the TEST or DEBUG privilege.
- The last n files of the PERM GROUP (where *n* is set by the NUMLK parameter) must correspond exactly to the last *n* files of the TEMP GROUP.

For more information on procedure locking and the NUMLK parameter, refer to the *Model 204 System Manager's Guide*.

Cross-Reference facility

The Cross-Reference facility is a Dictionary facility that can be invoked from both the Dictionary Main Menu and Model 204 command level. It produces reports for users who develop and maintain Model 204 User Language procedures.

The output reports show the line numbers where language elements such as labels, functions, images and variable names occur in a specified set of procedures. Elements within subroutines and nested INCLUDEs can also be cross-referenced.

The Cross-Reference Report is produced in batch mode (by a batch job in OS and DOS, by a service machine in CMS). Prior to submitting a cross-reference job, the user can specify:

- A set of procedures in a procedure file or group
- A set of language elements to be cross referenced
- Substitute values for User Language global dummy strings
- Job-related parameters such as output destination and lines per page

The Cross-Reference facility also includes Preview and Browse functions, which inform the user about the procedures selected for processing.

For a complete description of the Cross-Reference facility, refer to the *Model 204 Dictionary and Data Administration Guide*.

Part V
Data Integrity

Part V provides information on:

- How Model 204 resolves conflicts between concurrently running User Language requests
- Methods for recovering from hardware or software failures

24

Record Level Locking and Concurrency Control

In this chapter

Overview

This chapter discusses aspects of request design related to concurrency control in a multi-user environment and presents statements and Model 204 options that can be used to ensure logical consistency.

Concurrent updates

A user executing a User Language request expects that records retrieved will not be modified by another user's concurrent request until the user's own request has been completed. For example, consider these requests being run concurrently by two users:

User 1

```
BEGIN
BLUE.CARS: FIND ALL RECORDS FOR WHICH
               COLOR = BLUE
           END FIND
           FOR EACH RECORD IN BLUE.CARS
               PRINT MAKE AND COLOR
           END FOR
END
```

User 2

```
BEGIN
BLUE.BUICKS: FIND ALL RECORDS FOR WHICH
                 MAKE = BUICK AND COLOR = BLUE
             END FIND
             FOR EACH RECORD IN BLUE.BUICKS
                 CHANGE COLOR TO RED
              END FOR
 END
```

User 1 expects only the make and the color BLUE to be printed. User 2's request must be prevented from changing User 1's records before they are printed.

Record locking

The technique used by Model 204 to prevent overlapping updates is called record level locking. Conflicts arise when one or more users are reading a file and another user attempts to update the file or when two or more users attempt to perform file maintenance on the same records retrieved from that file.

Record level locking

Record locking modes

Model 204 performs record level locking in two modes:

Lock mode	Allows…
Share	One or more users to read a file. Any number of users can have shared control of a record or record set concurrently.
Exclusive	Single user to update the file. An exclusive lock is not compatible with other exclusive locks nor with any shared locks.

Request compilation and evaluation

Record-level locking is performed in the following manner. User Language requests are processed in two phases: compilation and evaluation. All requests can be compiled regardless of the operations to be performed or of other requests being compiled or evaluated at the same time. All requests are allowed to begin the evaluation stage.

Evaluation rules

When the evaluation involves records or sets of records, Model 204 automatically ensures that operations achieves the expected results by adhering to the following set of rules.

FIND statement

The FIND statement immediately locks the set of records it has retrieved in share mode. If the locking is successful, none of the records in that set can be updated by another user until the entire request (including request continuations) has been completed, or the records have been released by the RELEASE statement.

A FIND statement executed in a loop releases the old found set as soon as the statement is re-executed. The final set selected by the FIND remains locked until the end of the request. An unlabeled FIND statement, such as a FIND AND PRINT COUNT statement, does not lock records at all.

FIND WITHOUT LOCKS statement

The FIND WITHOUT LOCKS statement executes a FIND statement without obtaining any record locks. The found set of records is indistinguishable from a list (except that it is referenced with “IN label” syntax). The FIND WITHOUT LOCKS statement should be used with caution; logical inconsistencies might occur. See “FIND WITHOUT LOCKS statement” on page 24-6 for more information.

FOR EACH RECORD

The FOR EACH RECORD statement is handled as follows:

- If the FOR EACH RECORD statement refers to a found set (has the IN label clause), it retains the lock of the found set in share mode.
- If the FOR EACH RECORD statement refers to a list (has the ON LIST listname clause), no records are locked.
- If the FOR EACH RECORD statement does not refer to a previous found set or a list, all records in the current file or group are locked in share mode. After all the records are processed by the loop, the lock is dropped.

If the WHERE or WITH clause is used on the FOR EACH RECORD statement, only those records that satisfy the retrieval specifications are locked in share mode. After all the records are processed by the loop, the lock is dropped for all the processed records.

If the FOR EACH RECORD statement has the IN ORDER BY EACH clause, the lock is not released until all loop processing ends.

The lock on records retrieved as part of a FOR EACH RECORD statement is released as soon as the records are processed unless the record is changed and the file is a transaction backout file. For more information about transaction backout files, refer to “Transaction backout” on page 25-3.”

FOR RECORD NUMBER

The FOR RECORD NUMBER statement locks the specified record in share mode.

DELETE ALL RECORDS IN

The DELETE ALL RECORDS IN statement temporarily locks the set of records to be deleted in exclusive mode before deletion occurs. The locking does not succeed if another user has access to any of the records through a FIND or file maintenance statement. Once a record has been deleted, the exclusive lock on the record is released because the record no longer exists in the file.

ADD/CHANGE/DELETE, DELETE RECORD, and INSERT

The ADD, CHANGE, DELETE fieldname, DELETE RECORD, and INSERT statements all lock the current record in exclusive mode before updating it. Each of these statements must occur within a FOR EACH RECORD loop. The current record remains locked until it passes through the loop unless it has been deleted. If the record has been deleted, the exclusive lock on the record is released because the record no longer exists in the file.

STORE RECORD

The STORE RECORD statement tries to lock the newly created record in exclusive mode.

END MORE

If a request ends with END MORE, records found by that request remain locked and cannot be modified by other users. If the request ends with END, all records are released as soon as execution is completed.

Locking conflicts

If Model 204 cannot lock a record, a locking conflict occurs, which are discussed in detail in “Locking conflicts” on page 24-8.

FIND WITHOUT LOCKS statement

The FIND WITHOUT LOCKS statement executes a FIND statement without obtaining any record locks. The resulting found set is indistinguishable from a list (except it is referenced with IN *label* syntax).

Note: The FIND WITHOUT LOCKS feature should be used only to solve specific performance problems. Before using the FIND WITHOUT LOCKS statement, please take into account the “Usage notes” following “Syntax”.

Syntax

The format of the FIND WITHOUT LOCKS statement is:

```
{FIND WITHOUT LOCKS | FDWOL} [ALL] RECORDS
 [IN label | ON [LIST] listname]
 [FOR WHICH | WITH] retrieval conditions
```

Usage notes

Issues involved with using FIND WITHOUT LOCKS include:

- Logical integrity of data is at risk when another user:
  - Is in the midst of changing values which are related
  - Changes or deletes the field which caused the record to be found
- Physical integrity error messages or snaps are generated, including:
  - SICK RECORD messages are sent when extension records get deleted (the record isn’t really sick; it just temporarily appears that way to Model 204)
  - SICK RECORD messages being sent from FOR EACH OCCURRENCE (FEO) statements when the record is modified by another user (again, the record isn’t really sick)
  - NONEXISTENT RECORD messages are sent when entire records get deleted

Examples of appropriate use of the FIND WITHOUT LOCKS statement include:

- When there is one user at a time per record (for example, scratch records or bank teller applications where an account is usually modified by one teller at a time)
- Report programs in a heavy update environment

Examples of inappropriate uses of the FIND WITHOUT LOCKS, which can result in snaps, include:

- Report program in a heavy delete environment (results in many NONEXISTENT RECORD messages)
- Retrievals in which the selection criteria can be changed by other users

- Reuse Record Number files (except possibly scratch files keyed on the user ID)

Locking conflicts

A typical locking situation is as follows. The first user FINDs a large set of records and starts to print a long report. The second user FINDs some of the same records and tries to update them with a CHANGE statement.

Responses to locking conflicts

Under these circumstances, the second user must decide whether to cancel the request or try again. The user receives a message noting that the locking failed and then is queried: `DO YOU REALLY WANT TO TRY AGAIN`? The user can respond in one of three ways:

- Reply `N`, thereby cancelling the request.
- Reply `Y` and try to lock again immediately.
- Wait a minute or two and reply `Y`.

ENQRETRY parameter

The number of times a request automatically attempts to lock a record or set of records before notifying the user of a conflict is determined by the ENQRETRY parameter. Between attempts, Model 204 waits until the record or records that were held by another user are released or until three seconds pass. For more information on the ENQRETRY parameter, refer to the *Model 204 Command Reference Manual*.

ON RECORD LOCKING CONFLICT and ON FIND CONFLICT statements

In Model 204 you can specify the action to take if, after a record locking attempt, an effort to lock a set of records is still unsuccessful. The following statements specify the action to take in detail.

Statement	Can be used for…
ON RECORD LOCKING CONFLICT	Any type of conflict (including a retrieval statement conflict) that arises during a record locking attempt.
ON FIND CONFLICT	Only a conflict that arises during the evaluation of a FIND statement or a FOR EACH RECORD statement used for retrieval.

Syntax The format for these ON units is:

```
[label] ON {RECORD LOCKING CONFLICT | FIND CON-
FLICTS}
```

If both types of ON units are active

ON RECORD LOCKING CONFLICT and ON FIND CONFLICT follow the same rules as other ON units (see “ON units” on page 12-21). If both ON RECORD LOCKING CONFLICT and ON FIND CONFLICT are active within a request when a conflict occurs, the conflict is handled in the following manner:

- If the conflict results from a FIND statement or FOR EACH RECORD statement used for retrieval, the ON FIND CONFLICT unit is invoked.
- If the conflict results from a condition other than the FIND or FOR EACH RECORD statement used for retrieval, the ON RECORD LOCKING CONFLICT unit is invoked.

CLEAR ON statement

The definition of an ON RECORD LOCKING CONFLICT or ON FIND CONFLICT unit is cleared by the following statement:

```
CLEAR ON {RECORD LOCKING CONFLICT | FIND CON-
FLICTS}
```

After a CLEAR ON RECORD LOCKING CONFLICT or CLEAR ON FIND CONFLICT statement, a record locking conflict does not invoke the corresponding ON unit.

PAUSE statement

You can use the PAUSE statement to cause the request to wait a specified number of seconds and then to retry the statement that caused the evaluation of the ON unit. The PAUSE statement is a bumpable wait.

Syntax

The format of the PAUSE statement is:

```
PAUSE [n | %variable]
```

Where

- The value of *n* must be in the range 0-600, allowing a maximum pause of 10 minutes. If *n* is not specified or is specified as zero, processing stops and does not continue until the user enters a carriage return. If *n* is specified, processing continues automatically after *n* seconds.
- The *%variable* is interpreted as a numeric value representing the number of seconds to wait. A 600 second limit still applies to the hard coded number value. However, the %variable does not have a limit.

  An invalid %variable, for example, %a=’xxx’, is treated as if zero was specified and a read is issued.

Note: You should generally specify small values for *n* and issue a PAUSE only when no records are held by the request. If *n* is large, the request can seem to be hung.

Handling Parallel Query Option/204 record locking conflicts

If a client request cannot complete because of a record locking conflict on the server system, the server automatically tries again to lock the record or set of records. The server tries again until it succeeds or until it has tried as many times as the value of the *client thread* ENQRETRY parameter. The value of the ENQRETRY parameter that is specified on the server thread has no effect on the number of retries.

If ENQRETRY attempts to lock a record or set of records do not succeed, the server notifies the client about the conflict. If an ON RECORD LOCKING or ON FIND CONFLICT unit is active, the unit is invoked. Otherwise, the client receives a message that the locking failed, followed by a prompt asking if the client wants to try again.

If the client enters N, the request is canceled. If the client enters Y, the server repeats the locking attempt cycle, making as many as ENQRETRY attempts before prompting again.

Record locking and release statements

The following statements can place a lock on a set of records or remove the lock placed on records.

Note: To remove the lock placed on records, you can also use the “COMMIT statement” on page 24-15.

FIND AND RESERVE statement

A User Language request can lock records in exclusive mode by using the FIND AND RESERVE statement. However, because records are exclusively locked, concurrency is reduced.

Syntax

The basic format of the FIND AND RESERVE statement is:

```
FIND AND RESERVE [ALL] RECORDS FOR WHICH
```

Example

```
BEGIN
ON RECORD LOCKING CONFLICT
    PRINT ’RECORD LOCKING CONFLICT OCCURRED WITH ’ -
         WITH $RLCUSR
    PRINT ’FILE ’ WITH $RLCFILE
    PRINT ’RECORD ’ WITH $RLCREC
END ON

POL.HLDR:  IN CLIENTS FIND AND RESERVE ALL RECORDS -
               FOR WHICH POLICY NO = 100015
               RECTYPE = POLICYHOLDER
           END FIND
OWNER.POL: IN VEHICLES FIND AND RESERVE ALL RECORDS -
               FOR WHICH OWNER POLICY = 100015
           END FIND
           FOR EACH RECORD IN OWNER.POL
               %NEW.PREMIUM = VEHICLE PREMIUM + 100
               CHANGE VEHICLE PREMIUM TO %NEW.PREMIUM
               %TOTAL.PREMIUM = %TOTAL.PREMIUM + %NEW.PREMIUM
           END FOR
           FOR EACH RECORD IN POL.HLDR
               CHANGE TOTAL PREMIUM TO %TOTAL.PREMIUM
           END FOR
END
```

In the preceding example, the first FIND AND RESERVE statement prevents access to TOTAL PREMIUM while its corresponding VEHICLE PREMIUMs are being changed.

RELEASE RECORDS statement

Records found using the FIND AND RESERVE statement are held in exclusive status until the end of the request or until they are released explicitly by the user with the RELEASE RECORDS statement. The RELEASE RECORDS statement also can be used to release records from SORT statements or records obtained in share status by FIND statements issued in the regular form. Records released in this manner are no longer available to the request and might have to be found again. The original FIND set or list from which the sorted set was built is not affected.

Note: To avoid confusing results, CCA recommends that you issue RELEASE RECORDS at the end of a loop and not in the middle of one, since the statement relinquishes control of a found set.

Syntax The format of the RELEASE RECORDS statement is:

```
RELEASE RECORDS {IN label | ON [LIST] listname}
```

where *label* is the statement label of the FIND statement that locked the records.

The RELEASE RECORDS statement is supported in remote file and scattered group contexts.

Processing

The RELEASE RECORDS statement releases the lock and empties the found set. When the RELEASE statement refers to a SORT statement, the space occupied by the temporary sorted record copies is released.

RELEASE RECORDS ON is equivalent to the CLEAR LIST statement.

RELEASE ALL RECORDS statement

Description A RELEASE ALL RECORDS statement terminates the lock in share mode placed on records by the FIND statement. RELEASE ALL RECORDS statement also clears all lists and the results of all SORT statements, and sets the current record number to -1.

After a RELEASE ALL RECORDS statement is processed in a FOR EACH RECORD loop, subsequent references to the current record in the loop cannot find a current record. If processing returns to the top of the loop, no new record is available, and the first statement after the end of the loop is executed.

Syntax The format of this release statement is:

```
RELEASE ALL RECORDS
```

The RELEASE ALL RECORDS statement is supported in remote file and scattered group contexts.

RELEASE and COMMIT RELEASE statements with global foundsets and lists

The RELEASE statements and the COMMIT RELEASE statements empty the contents of a global found set, global sort set, or global list. The label and positions associated with a found set, sort set, or the list is still considered global, but it is empty. Global positions are not cleared by RELEASE or COMMIT RELEASE statements, however, without records there is nothing to process.

Lock pending updates

Model 204 provides a special facility, lock pending updates, that prevents updated records in one transaction (a sequence of file updating operations) from being used by other applications until the transaction ends. Lock pending updates ensures logical consistency without requiring the use of the FIND AND RESERVE statement.

Processing

If the lock pending updates option is specified, records are locked in share mode by a FIND statement. The first update to a record locks the record in exclusive mode and adds it to a set of updated locked records called the pending update pool. The record is not released from this exclusive lock at the end of the FOR EACH RECORD loop. Instead, the record is locked until the end of the transaction when the entire pending update pool is released.

Set with the FOPT parameter

Lock pending updates is an option of the FOPT parameter that is enabled or disabled on a file-by-file basis. Refer to the *Model 204 Command Reference Manual* for more information on the FOPT parameter.

COMMIT statement

The COMMIT statement ends the current transaction, dequeues checkpoints, and, for files with lock pending updates, releases the exclusive lock on updated records. The exclusive lock on the current record from a file with the REUSE RECORD NUMBER option disabled can be released only at the end of the FOR EACH RECORD loop.

The COMMIT statement is supported in remote context. If records on any remote nodes are updated, the COMMIT statement saves all remote updates on all remote nodes. Also, remote updates are committed automatically by the system when appropriate.

Syntax

The format of the COMMIT statement is:

```
COMMIT [RELEASE]
```

Example

```
BEGIN
GET.REC: FIND ALL RECORDS FOR WHICH
             LNAME = NELSON
         END FIND
         FOR EACH RECORD IN GET.REC
             ADD MONTH = NOV
             COMMIT
            PRINT FNAME AND LNAME AND EACH MONTH
         END FOR
NAME.CT: COUNT RECORDS IN GET.REC
         PRINT COUNT IN NAME.CT WITH ’ RECORDS
UPDATED’
END
```

The COMMIT statement inside the FOR EACH RECORD loop ends the current update unit each time the loop is processed. This results in several short update units instead of one long update unit. Record locking conflicts can be minimized by the use of frequent COMMIT statements.

RELEASE option

The RELEASE option of the COMMIT statement performs all the operations of the COMMIT and RELEASE ALL RECORDS statements. :

After processing a COMMIT RELEASE statement …	Any subsequent reference to the current…
FOR EACH RECORD loop	Record in the loop cannot find a current record. If processing returns to the top of the loop, no new record is available; the first statement after the end of each loop is executed.

After processing a COMMIT RELEASE statement …	**Any subsequent reference to the current…**
FOR EACH VALUE loop	Value still obtains the last value processed.

Usage To avoid confusing results, it is recommended that COMMIT RELEASE be issued at the end of a loop and not in the middle of one.

RELEASE and COMMIT RELEASE statements with global foundsets and lists

The RELEASE statements and the COMMIT RELEASE statements empty the contents of a global found set, global sort set, or global list. The label and positions associated with a found set, sort set, or the list is still considered global, but it is empty. Global positions are not cleared by RELEASE or COMMIT RELEASE statements, however, without records there is nothing to process.

25

Data Recovery

In this chapter

Overview

If hardware or software failures occur during the process of updating Model 204 files, database integrity problems can arise. These problems result from the fact that updates to a file can require coordinated changes, and a failure can interrupt the process of writing to storage a complete set of these changes.

Normally in a Model 204 installation, the system manager is responsible for the coordination of recovery activity for the installation, and the file manager determines which recovery features are applied to each of the files.

For a complete description of the Model 204 recovery process, refer to the *Model 204 File Manager's Guide*.

Transaction backout

One of the recovery features that can be selected by the file manager is an integrity facility called transaction backout. This facility can reverse the effects of an incomplete transaction (a sequence of file updating operations).

Application considerations

Although the file manager determines whether the transaction backout facility is defined for a file, the application designer must understand the facility's capabilities and requirements in order to ensure file integrity for an application.

This chapter explains those capabilities and requirements and describes how applications can be designed to ensure file integrity. In particular, see the last section in this chapter for application design considerations.

Transaction backout

The transaction backout facility allows Model 204 to logically reverse the effects of an incomplete update to a file.

A backout can be initiated only on incomplete update units; completed update units cannot be backed out. A backout can be automatically performed by Model 204 or initiated by the user.

FOPT and FRCVOPT parameters

Transaction backout is an option of the FOPT and FRCVOPT parameters and is enabled or disabled on a file-by-file basis. Refer to the *Model 204 Command Reference Manual* for more information on these parameters.

Types of backout

You can use the transaction backout facility to specify two types of files:

Backout file types	**File options set to…**	**Discussion**
Transaction backout	Enable both lock pending updates and backout.	A transaction can be backed out automatically by Model 204 or manually by the application. A transaction backout file ensures: • Logical consistency • Data and file integrity, • High degree of data sharing.
Non-transaction backout	Disable transaction backout.	Incomplete updates cannot be automatically reversed. Non-transaction backout files cannot be accessed remotely. Attempts to open a non-transaction backout file in remote context fails and generates the following error message: `M204.1973: NON-TBO REMOTE FILE`

Update units

An ***update unit*** is any sequence of operations that updates the database and that has a beginning and ending point. One update unit must end before another update unit can begin.

A request or procedure can have two types of update units:

Update unit type	Discussion
Backoutable	A backoutable unit consists of updates to transaction backout files using file updating statements (such as STORE RECORD). The unit can either be completed so that it persists in the file, or backed out so that the update is logically reversed in the file.
Non-backoutable	A non-backoutable unit consists of updates to non-transaction backout files.

Backoutable units

Beginning a backoutable unit

A backoutable unit begins at the execution of the first User Language statement that performs an update operation on a transaction backout file. The statements that perform update operations are:

- ADD *fieldname = value*
- CHANGE *fieldname* TO *value*
- DELETE EACH *fieldname*
- DELETE *fieldname* [= *value*]
- DELETE RECORD
- DELETE RECORDS
- FILE RECORDS
- INSERT *fieldname = value*
- STORE RECORD

Ending a backoutable unit

A backoutable unit ends when the unit is committed by using the COMMIT statement or is backed out. Other conditions (such as an END statement in a request or the BACKOUT statement discussed later in this chapter) also can end an active update unit. Refer to the *Model 204 File Manager's Guide* for a complete list of conditions that end an active update unit.

Non-backoutable units

Beginning a non-backoutable unit

A non-backoutable unit begins with the first User Language statement that performs an update operation on a non-transaction backout file.

Ending a non-backoutable unit

A non-backoutable unit ends when the unit is committed by using the COMMIT statement. Other conditions (such as an END statement in a request or the BACKOUT statement discussed later in this chapter) also can end an active update unit. Refer to the *Model 204 File Manager's Guide* for a complete list of conditions that start or end an active update unit.

Using backout

The transaction backout facility performs two types of backouts:

Backout type	Invoked…
Automatic	• Automatically by a request cancellation • An attention or *CANCEL command without an ON ATTENTION unit • File full condition • User restart.
Manual	Using the BACKOUT statement within a request.

Automatic backout

Model 204 automatically backs out the current update unit for a transaction backout file under any of the following conditions:

- If a request is cancelled by Model 204
- If there is a file integrity problem
- If Model 204 is restarting a user who has an active update unit

When a request accessing a transaction backout file is cancelled, the following events occur:

- The user receives a message that explains why the request was cancelled.
- The current update unit is backed out automatically.
- Model 204 returns to the terminal command level unless an ON ERROR or ON ATTENTION unit is invoked.

Remote updates are backed out automatically when appropriate.

Manual backout

You can back out an incomplete update unit by using the BACKOUT statement. This statement is valid only for updates to transaction backout files.

The BACKOUT statement releases the exclusive lock on updated records and backs out the current update unit. No found sets are released and the current record does not change.

After the BACKOUT statement is processed, evaluation continues with the next statement. A message is displayed to the user upon the successful completion of a backout operation.

The BACKOUT statement is supported in remote context. If records on any remote nodes have been updated, the BACKOUT statement causes all remote updates on all remote nodes to be backed out, in addition to any local updates.

Example The BACKOUT statement is typically used in data entry applications where a transaction might have to be backed out after it has updated a file (for example, a sales or airline reservations application).

An example of a request that uses the BACKOUT statement is provided below.

```
BEGIN
*
* DECREMENT INVENTORY BEFORE MAKING SALE TO CUSTOMER
*
DECLARE %INPUT STRING LEN 70
DECLARE %VINTNER STRING LEN 30
DECLARE %SELECTION STRING LEN 40
*
* ENTER PURCHASER’S SELECTION
*
GETVAL:     %INPUT=$READ(’ENTER VINTNER/SELECTION:’)
            IF %INPUT EQ ’QUIT’ THEN
                STOP
            END IF
            %VINTNER=$SUBSTR(%INPUT,1,$INDEX(%INPUT,’/’)-1)
            %SELECTION=$SUBSTR(%INPUT,$INDEX(%INPUT,’/’)+1)
FDVINTNER: FD REC=VINTNER AND NAME=%VINTNER
CTVINTNER: COUNT RECORDS IN FDVINTNER
            IF COUNT IN CTVINTNER EQ ’0’ THEN
                PRINT ’VINTNER DOES NOT EXIST TRY AGAIN’
                JUMP TO GETVAL
            END IF
*
* IF SELECTION EXISTS THEN UPDATE RECORD BY DECREASING
* THE AMOUNT ON HAND
*
CTSELECTION: FOR EACH RECORD IN FDVINTNER
FEOSELECT:   FOR EACH OCCURRENCE OF SELECTION
               IF SELECTION(OCCURRENCE IN FEOSELECT) EQ -
                 %SELECTION THEN
                 %ONHAND=ONHAND(OCCURRENCE IN FEOSELECT)
                     %DECREMENT=%ONHAND-1
                     CHANGE ONHAND(OCCURRENCE IN FEOSELECT)-
                         TO %DECREMENT
                     JUMP TO GETCASH
                 END IF
             END FOR
             PRINT ’SELECTION DOES NOT EXIST TRY AGAIN’
             JUMP TO GETVAL
             END FOR

*
* IF CUSTOMER IS READY TO PROCEED WITH THE SALE, THEN
* COMMIT THE TRANSACTION. IF NOT, THEN BACKOUT RECORD
* UPDATED
*
GETCASH: %CASH=$READ(’DID YOU GET CASH/CREDIT CARD FROM -
               CUSTOMER Y/N:’)
```

```
         IF %CASH EQ 'Y' THEN
             COMMIT
             JUMP TO FINISH
         ELSE
             BACKOUT
             PRINT 'INVENTORY WAS NOT UPDATED '
             PRINT 'PURCHASE STOPPED'
             JUMP TO DONE
         END IF
FINISH:  PRINT 'TRANSACTION COMPLETED'
         PRINT 'THANK YOU'
DONE:    *COMMENT
END    *
```

Design considerations for transaction backout files

This section identifies the factors to consider when designing an application that uses transaction backout files.

Transaction backout provides increased file logical consistency over the FIND share lock and FOR loop exclusive lock, and provides increased data sharing over the FIND AND RESERVE exclusive lock. However, effective use of the transaction backout facility depends on careful design and implementation.

Update requests

Requests that update transaction backout files cannot access non-transaction backout files in any way. This restriction ensures that all update units for a transaction backout file are logged correctly, can be backed out, and do not overlap with any update units that cannot be backed out.

Non-updating requests can access any type of file, and requests that update non-transaction backout files can read transaction backout files.

ON ATTENTION units

ON ATTENTION units should be used within requests to avoid inadvertently backing out an active update unit. If an ON ATTENTION unit is not specified and the user presses one of the ATTENTION identifier (AID) keys at the terminal or enters `*CANCEL` at a terminal I/O point, then the request is cancelled. The cancellation causes an automatic backout of the current update unit if the request updates transaction backout files.

CCATEMP space

The backout facility must have the necessary information to back out each active update unit in the Model 204 run. A log of compensating updates is built for each active unit on the system file, CCATEMP. When the unit ends or is backed out, the log for that particular unit is discarded.

To minimize the amount of CCATEMP space used, update units should be designed to contain only a few file updates. Units of a sizable amount can greatly increase the amount of CCATEMP space used by Model 204. Therefore, update units should be committed frequently to minimize the size of the backout log.

Logical inconsistency

Although transaction backout does increase the logical consistency of a file through the lock pending updates option, logical inconsistency can occur when an update unit is backed out. For example, logical inconsistency can arise when a unit involving DELETE RECORD is backed out because deleted records are not locked by the lock pending updates option.

Example The following requests illustrate such a logical inconsistency:

```
User 1                              User 2

BEGIN
SMH: FIND ALL RECORDS FOR WHICH
         NAME = SMITH
     END FIND
     FOR EACH RECORD IN SMH
         DELETE RECORD
     END FOR
JNS: FIND ALL RECORD FOR WHICH      BEGIN
         NAME = JONES               FD.RECS:  FIND ALL RECORDS FOR
                                    WHICH
     END FIND                                     NAME = SOLOBY OR -
     FOR EACH RECORD IN JNS                       NAME = SMITH OR -
         ADD TYPE = SPECIAL                       NAME = SAUNDERS
             .                                END FIND
             .                               FOR EACH RECORD IN FD.RECS
             .                                    ADD TYPE = SPECIAL
             .                                END FOR
             .                      FD.SPEC:  FIND ALL RECORDS FOR
                                    WHICH
             .                                    TYPE = SPECIAL
             .                                END FIND
             .                      PRT:     FOR EACH RECORD IN FD.SPEC
             .                                    PRINT NAME AND TYPE
             .                                END FOR
DEL: BACKOUT                        END
ALL: FIND ALL RECORDS
     END FIND
     FOR EACH RECORD IN ALL
         PRINT NAME AND TYPE
     END FOR
END
```

User 1 deletes the set of records with NAME = SMITH. Although the update unit is not complete, the deleted set of records is not locked. Before user 1’s unit completes, user 2 finds the set of records with NAME = either SOLOBY, SMITH, or SAUNDERS. There is no locking conflict because the deleted records are not locked. User 2 adds a field to the found set, prints a report, and ends.

User 1 then decides to back out the unit in progress. The deletion of the SMITH records is in the backed out unit and the SMITH records reappear on the file. However, because the SMITH records do not have the TYPE = SPECIAL field added by user 2, there is a logical inconsistency. The chances of this type of inconsistency can be reduced by keeping units short, especially when the units involve record deletion. If user 1 had a COMMIT statement after the DELETE RECORD loop, no inconsistency would arise unless the backout was automatically activated during that short piece of the user request. It also is likely that the records deleted by an application are meant to be deleted, and a logical inconsistency in a set of records intended for deletion is unimportant.

FILE RECORDS statement

Similar logical inconsistency can occur using the FILE RECORDS statement because FILE RECORDS does not lock any records. For the FILE RECORDS statement, a FIND statement can be used to lock the records to be updated in share mode, or FIND AND RESERVE can be used to lock the records in exclusive mode. If the FIND set is locked exclusively and not released until after the update unit containing the FILE RECORDS statement has ended, then logical inconsistency is prevented.

Terminal I/O points

Terminal I/O points should not be placed between the start of an update unit and the unit’s end or backout. This precaution stems from the exclusive lock placed on updated records. If a response is required from a terminal operator either to complete or back out an update unit, there is a possibility that a set of records could be locked for an extended period of time.

Part VI
Reference and Appendix

Part VI includes reference chapters and an appendix covering:

- Commands and statements
- Functions
- Abbreviations
- Reserved words and characters
- Conversion, rounding, and precision rules
- Field attributes
- Obsolete features

26

Command and Statement Syntax

In this chapter

Overview

This chapter summarizes User Language syntax and conventions, many of which are also discussed throughout this manual.

The statements are listed in alphabetical order. Later sections in this chapter provide other User Language syntax information.

All of the User Language statements listed can be used between a BEGIN (or MORE) command and an END (or END MORE) statement.

All system control commands are presented in the *Model 204 Command Reference Manual*.

Notation conventions used in this chapter

In addition to the standard notation conventions listed in the Preface of this manual, this chapter uses the following syntax notation conventions:

Syntax notation	**Indicates that…**
Single asterisk (*)	Statements can be preceded by an IN clause, if there is no reference to a previous set (label or list). See “IN clause syntax” on page 26-26 for more discussion.
Two asterisks (**)	Construct can appear only within a record loop.
Plus sign (+)	Construct requires the optional Horizon feature.
Two plus signs (++)	Construct requires the optional User Language to Database 2 feature.
C	The syntax applies to a Model 204 command as well as a User Language statement, except any %variable options or clauses. Model 204 commands are listed alphabetically and documented in *Model 204 Command Reference Manual*.
Lower case italic	Constructs are replaced with variable information.

- A field name (%%) variable can be used anywhere *fieldname* appears. The %%variable can contain its own subscript, separate from the field name subscript.

  Note: The subscript of an array element must be specified before a field name subscript.

- The lower case constructs—*retrieval-conditions, print-specifications, expression, subscript, attribute,* and *type*—are discussed separately following the syntax summaries, beginning with “Value specification syntax” on page 26-21.

User Language statements

** ADD *fieldname = value*

ARRAY [*arrayname*] OCCURS {*n | UNKNOWN*}
 DEPENDING ON {*itemname* | *%variable*}
 [AFTER {*itemname* | *arrayname*}
 | AT {*position* | *itemname* | *imagename1* | *arrayname*}]

AUDIT *print-specifications*

C BACKOUT

BYPASS [PENDING STATEMENT]

CALL {*label* | *subname*
 [([*expression* | *%variable* | [LIST] *listname*] [,…])]}

** CHANGE *fieldname* [(*subscript*)] = *value1* TO *value2*

CLEAR
 {[[<u>ALL</u> | TEMP | LISTFDST | POSITION] [GLOBAL]]
 OBJECTS
 | GLOBALS
 | GLOBAL {IMAGE | SCREEN | MENU | LIST | FOUNDSET
 | POSITION [<u>PERM</u> | TEMP]}
 {'*objectname*' | *%variable*}}

```
 *  CLEAR LIST listname
    ____________________________________________________
    CLEAR ON
     {ATTENTION | ERROR | FIELD CONSTRAINT CONFLICT
     | FIND CONFLICT | RECORD LOCKING CONFLICT
     | MISSING FILE | MISSING MEMBER}
    ____________________________________________________
    CLEAR TAG {screenname | %screenname:inputname}
    ____________________________________________________
    CLOSE
     {DATASET {ext-filename | %variable}
     | [EXTERNAL] {ext-filename | TERMINAL | %variable}}
    ____________________________________________________
+C  CLOSE PROCESS
     {cid | processname | %variable}
     [SYNCLEVEL | FLUSH | %variable]
    ____________________________________________________
    CLOSE PROCESS
     {cid | processname | %variable}
     [SYNCLEVEL | CONFIRM | FLUSH | ERROR | %variable]
    ____________________________________________________
    COMMIT [RELEASE]
    ____________________________________________________
 +  CONFIRM {cid | processname | %variable}
            REQSEND %variable
    ____________________________________________________
 +  CONFIRMED {cid | processname | %variable}
    ____________________________________________________
    CONTINUE
```

```
** COUNT OCCURRENCES OF fieldname
```

```
COUNT RECORDS {IN label | ON [LIST] listname}
```

```
[DECLARE] declaration
```

where *declaration* is one of the following:

```
 LABEL labelname [GLOBAL | COMMON]
 LIST listname [IN [FILE | [PERM | TEMP]
      GROUP]] name] [GLOBAL | COMMON]
 IMAGE imagename [AT {itemname | imagename1
       | arrayname}
       | GLOBAL [PERMANENT | TEMPORARY]
       | [PERMANENT | TEMPORARY] GLOBAL
       | COMMON]
 MENU menuname [GLOBAL [PERMANENT | TEMPORARY]
      | [PERMANENT | TEMPORARY] GLOBAL
      | COMMON]
 SCREEN screenname [GLOBAL [PERMANENT | TEMPORARY]
        | [PERMANENT | TEMPORARY] GLOBAL
        | COMMON]
 %variable [IS] {FIXED [DP n] | FLOAT}
           [ARRAY (d1 [,d2 [,d3]]])]
           [COMMON] [INITIAL] [STATIC]
 %variable [IS] STRING [LEN n] [DP {n | *}]
           [ARRAY (d1 [,d2[,d3]]])]
           [NO FIELD SAVE] [COMMON]
           [INITIAL] [STATIC]
 SUBROUTINE subname
```

```
    [(type [INPUT | OUTPUT | INPUT OUTPUT] [,…])]
```

type is one of the following:

– Scalar %variable of the following format:

```
{STRING [LEN] n [DP {n | *}] | [FIXED [DP n]
| FLOAT]}
```

– Array %variable of the following format:

```
{STRING [LEN n] [DP [n | *}] [ARRAY (* [,*[,*]])
 [NO FIELD SAVE]]
 | [FIXED [DP n] | FLOAT] [ARRAY (* [,*[,*]])]}
```

– A list of records of the following format:

```
[LIST] [IN {FILE | [PERM | TEMP] GROUP} name]
```

```
DEFAULT CURSOR [READ | REREAD | PRINT]
              {ITEMID n | itemname | ROW n COLUMN m}
```

```
DEFAULT {TITLE
        | PROMPT
        | INPUT [DEBLANK | NODEBLANK] [PAD WITH 'c']
          [LEN m [DP [k | *}]] [UPCASE | NOCASE]
        | [TAG [attributes] [WITH 'c']]}
        [[READ] attributes]
        [REREAD attributes]
        [PRINT attributes]
```

```
DEFAULT SKIP n POSITIONS[S]
             [INITIAL {NULL | character | BLANK}]
```

```
DEFAULT [TYPE]
 {STRING | BINARY | PACKED | ZONED | FLOAT | EFORMAT}
 {LEN {n | UNKNOWN} | BITS n | DIGITS n}
 [DP {k | *}] [BP n]
 [SIGNED | UNSIGNED] [PAD {BLANK | character | NULL}]
```

[STRIP | NOSTRIP] [JUSTIFY {LEFT | RIGHT}]

[INITIAL {BLANK | ZERO | NULL | *literal*}]

** DELETE *fieldname* [(*subscript*)] [= *value*]

** DELETE EACH *fieldname*

** DELETE RECORD

DELETE [ALL] RECORDS {IN *label* | ON [LIST] *listname*}

END BLOCK *label*

END {FIND | IF | FOR | ON | REPEAT | STORE

| SUBROUTINE} [*label*]

END {ARRAY | IMAGE | MENU | SCREEN}

END [MORE | NORUN | USE]…

END UPDATE

FILE RECORDS {IN *label* | ON [LIST] *listname*}

UNDER *fieldname = value*

---

* FIND [AND RESERVE] [ALL] RECORDS

[IN *label* | ON [LIST] *listname*]

[FOR WHICH | WITH] *retrieval conditions*

---

* FIND [ALL] VALUES OF *fieldname*

[FROM *value1*] [TO *value2*]

[[NOT] LIKE *'pattern'*]

---

* FIND AND PRINT COUNT [*retrieval-conditions*]

---

* {FIND WITHOUT LOCKS | FDWOL}

[ALL] RECORDS [IN *label* | ON [LIST] *listname*]

[FOR WHICH | WITH] *retrieval-conditions*

FLUSH PROCESS {*cid* | *processname* | %*variable*}

** FOR {EACH | *k*} {OCCURRENCE | OCCURRENCES} OF *field-name*

---

* FOR {EACH | *k*} {RECORD | RECORDS}

[IN *label* | ON [LIST] *listname*]

IN [ASCENDING | DESCENDING]

[SORTKEY] ORDER [BY [EACH] *fieldname*]

[FROM *value1*] [TO *value2*] [BY {%*variable* | *literal*}]

[**OPT**IMIZING **FNV**]

[{WHERE | WITH} *retrieval-conditions*]

---

* FOR {EACH | *k*} {VALUE | VALUES} OF *fieldname*

[FROM *value1*] [TO *value2*] [[NOT] LIKE *pattern*]

[IN [ASCENDING | DESCENDING] [CHARACTER | **NUMERIC**AL]

[RIGHT-ADJUSTED] ORDER]

____

FOR {EACH | *k*} {VALUE | VALUES} IN *label*

____

* FOR RECORD NUMBER *value*

____

FOR RECORD NUMBER {*value* | IN *label*} [**OPT**IMIZING **FNV**]

____

FOR %*variable* FROM *expression1*

{TO *expression2* [BY *expression3*]

| [BY *expression3*] TO *expression2*}

Note: The BY clause, when omitted, defaults to a value of 1.

____

IDENTIFY

{[IMAGE] *imagename*

| %*imagename*:*itemname* LEN {*n* | %*variable*}

| %*imagename*:*arrayname* OCCURS {*n* | %*variable*}}

____

C IF *expression* THEN *statements*

[ELSE *statements* | ELSEIF *expression* THEN *state-ments*]

____

[DECLARE] IMAGE *imagename*

[GLOBAL [**PERM**ANENT | **TEMP**ORARY]

| [**PERM**ANENT | **TEMP**ORARY] GLOBAL

| COMMON [AT [*itemname* | *imagename1* |*arrayname*}]]

____

* INCLUDE *procedurename*

____

INPUT *inputname* [AT [COLUMN] *n*]

```
  [TO [COLUMN] m | [LEN m] DP {k | *}]]
  [UPCASE | NOCASE] [DEFAULT 'value']
  [DEBLANK | NODEBLANK] [PAD WITH 'c']
  [REQUIRED | ALPHA | ALPHANUM | MUSTFILL
  | ONEOF literal [,literal]…
  | [NUMERIC] [RANGE lo [TO] hi [AND lo [TO] hi] …]
  | VERIFY 'characters']…
  [[READ] attributes] [REREAD attributes]
  [PRINT attributes]
  {TAG [attributes] [WITH 'c']] [ITEMID n]
____________________________________________________
INSERT fieldname [(subscript)] = value
____________________________________________________
+ INVITE {cid | processname | %variable}
  [SYNCLEVEL | FLUSH | CONFIRM]
____________________________________________________
itemname IS [TYPE]
  {STRING | BINARY | PACKED | ZONED | FLOAT | EFORMAT}
  {LEN {n | UNKNOWN} | BITS n | DIGITS n
  | TO position}
  [DP {k | *}] [BP n] [SIGNED | UNSIGNED]
  [PAD {BLANK | character | NULL}] [STRIP | NOSTRIP]
  [ALIGN]
  [JUSTIFY {LEFT | RIGHT}]
  [INITIAL {BLANK | ZERO | NULL | value}]
  [AFTER {itemname | arrayname}
  | AT {position | itemname | imagename1 | arrayname}]
  [OCCURS {n [DEPENDING ON {itemname | %variable}]
  | UNKNOWN}]
```

JUMP TO *label*

JUMP TO (*label1* [,*label2*] …) *expression*

LOOP END

MAX PFKEY *n*

[DECLARE] MENU *menuname*
 [GLOBAL [**PERM**ANENT | **TEMP**ORARY]
 | [**PERM**ANENT | **TEMP**ORARY] GLOBAL
 | COMMON]

MODIFY {%*menuname*:*itemname* | %*screenname*:*itemname*}
 [TO] *attributes* [[FOR] {ALL | READ | REREAD | TAB
 | PRINT}]

C NEW PAGE

****** NOTE {*fieldname* [(*subscript*)]}

NOTE *'string'*

ON {ATTENTION | ERROR | FIELD CONSTRAINT CONFLICT
   | FIND CONFLICT | MISSING FILE | MISSING MEMBER
   | RECORD LOCKING CONFLICT} *statements*

OPEN {[DATASET | EXTERNAL] {*ext-filename*

```
     | %variable}
     | [EXTERNAL] {TERMINAL | %variable}}
     FOR {INPUT [OUTPUT] | OUTPUT [INPUT] | INOUT}
     [PASSWORD {value | %variable}]
```

```
C  OPEN [C] [[PERM | TEMP] GROUP | FILE]
    {name [AT {location | %variable | =}] |%variable}
    [PASSWORD {value | %variable}]
```

```
   OPEN PROCESS {processname | %variable}
    [CID {name | %variable}]
```

```
+  OPEN PROCESS {processname | %variable}
    [CID {name | %variable}]
    {outbound options | inbound options}
```

where *outbound options* are:

```
    [AT DESTINATION] [WITH] [USERID {%variable
    | 'string'}]
    [PASSWORD {%variable | 'string'}]
    [{ACCOUNT | PROFILE {%variable | 'string'}]
    [INITIAL {DATA 'string' | DATA %variable
    | IMAGE imagename] …]
```

and *inbound options* are:

```
    ACCEPT [INITIAL {DATA %variable | IMAGE image} …]
```

```
   PAUSE [n | %variable]
```

```
** PLACE RECORD ON [LIST] listname
```

```
PLACE RECORDS {IN label | ON [LIST] listname1}
 ON [LIST] listname2
```

____

```
POSITION {FOUNDSET foundsortset_name | LIST
list_name}
  [AT] position_name
```

Note: Use this form of the POSITION statement with FOR loop processing; see “POSITION statement” on page 20-20.

____

```
POSITION {ext-filename | %variable}
 AT KEY operator {value | %variable}
```

Note: Use this form of the POSITION statement with external file processing using images; see “POSITION statement” on page 17-42.

____

```
PREPARE
 {[IMAGE] imagename | [MENU] menuname
 | [SCREEN] screenname}
```

____

```
PRINT print specifications
```

____

**

```
{PAI | PRINT ALL INFORMATION}
 INTO array1, array2 [FROM start] [COUNT ct]
```

____

```
PRINT [MENU] menuname [ALERT]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

____

```
PRINT SCREEN screenname [ALERT] [[WITH] CURSOR]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

```
PROMPT {'text' | promptname} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [DP {k | *}]] [DEFAULT
'value']
 [[READ] attributes] [REREAD attributes]
 [PRINT attributes] [ITEMID n]
```

```
+ QUERY PROCESS {cid | processname | %variable} options
```

where *options* must be one or more of the following:

```
 STATE %variable
 PROCESSGROUP %variable
 REMOTEID %variable
 SYNCLEVEL %variable
 MODENAME %variable
```

```
READ [IMAGE] imagename
 FROM {ext-filename | TERMINAL | %variable}
 [PROMPT {'text' | %variable}]
 [NEXT | KEY operator {value | %variable}]
```

```
READ [MENU] menuname [ALERT]
 [TITLE ['text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

```
READ [SCREEN] screenname [ALERT] [NO REREAD] [[WITH]
 CURSOR]
 [TITLE {'text' | %variable} [AT [COLUMN] n]
 [TO [COLUMN] m | LEN m] [attributes]]
```

```
RECEIVE {IMAGE imagename | %variable}
```

FROM {*cid* | *processname* | %*variable*} [RESULT %*variable*]

+ RECEIVE {IMAGE *imagename* | %*variable*}

FROM {*cid* | *processname* | %*variable*}

RESULT %*variable*

RELEASE ALL RECORDS

RELEASE POSITION {*ext-filename* | %*variable*}

RELEASE RECORDS {IN *label* | ON [LIST] *listname*}

** REMEMBER [GLOBAL] *position_name*

[IN f*oundsortset_name* | ON *list_name*]

** REMOVE RECORD FROM [LIST] *listname*

REMOVE RECORDS {IN *label* | ON [LIST] *listname*}

FROM [LIST] *listname2*

REPEAT [FOREVER | *n* TIMES | WHILE *expression*]

REREAD [SCREEN] *screenname* [ALERT] [[WITH] CURSOR]

[TITLE {*'text'* | %*variable*} [AT [COLUMN] *n*]

[TO [COLUMN] *m* | LEN *m*] [*attributes*]]

RESET {HEADER | TRAILER} *m*

RETRY [PENDING STATEMENT]

RETURN

[DECLARE] SCREEN *screenname*
 [GLOBAL [**PERM**ANENT | **TEMP**ORARY]
 | [**PERM**ANENT | **TEMP**ORARY] GLOBAL
 | COMMON]

SEND {IMAGE *imagename* | *'string'* | %*variable*}
 TO {*cid* | *processname* | %*variable*}
 [REQSEND %*variable*]
 [FLUSH]

+ SEND {IMAGE *imagename* | *'string'* | %*variable*}
 TO {*cid* | *processname* | %*variable*}
 [FLUSH | CONFIRM]
 [REQSEND %*variable*]

+ SEND ERROR TO {*cid* | *processname* | %*variabl*e}
           REQSEND %*variable*

SET {HEADER | TRAILER} *m print-specifications*

SIGNAL PROCESS
 {*cid* | [*processname* | %*variable*} {*nnn* | %*variable*}

+ SIGNAL PROCESS

{*cid* | [*processname* | %*variable*}

SKIP *n* LINE[S]

[*itemname* IS] SKIP *n* POSITION[S]

[INITIAL {NULL | *character* | BLANK}]

SORT [*k*] RECORDS {IN *label* | ON [LIST] *listname*}

BY *key* [AND *key*] …

where:

key = *fieldname*

[VALUE {[ASCENDING | DESCENDING]

[CHARACTER | **NUMERIC**AL] | RIGHT-ADJUSTED]} …]

SORT [*k*] RECORD KEYS

{IN *label* | ON [LIST] *listname*}

BY *key* [AND *key*] …

where:

key = *fieldname*

[VALUE {[ASCENDING | DESCENDING]

[CHARACTER | **NUMERIC**AL] | [RIGHT-ADJUSTED]} …]

SORT VALUE IN *label* [IN [ASCENDING | DESCENDING]

[CHARACTER | **NUMERIC**AL] | [RIGHT-ADJUSTED] ORDER]

STOP [IF COUNT IN *label* EXCEEDS *n*]

---

*

STORE RECORD [*sort or hash key value*]

[*fieldname = value*]

•

•

```
SUBROUTINE
```

```
SUBROUTINE subname
 [(formal-parameter [INPUT | OUTPUT | INPUT OUTPUT]
 [,…])]
```

where *formal parameter* is one of the following:

```
 %variable [IS STRING [LEN n] [DP {n | *}]
           [ARRAY (*[,*[,*]]) [NO FS]]
           | IS {FIXED [DP n] | FLOAT}
           [ARRAY (*[,*[,*]])]]
 LIST listname [IN [FILE | [PERM | TEMP] GROUP] name]
```

```
TAG %screenname:inputname [attributes] [WITH 'c']
```

```
+ TEST [FOR]
   {ANY RECEIPT RETURN %variable
   | RECEIPT {cid | processname | %variable}}
```

```
TITLE {'text' | promptname}
 [AT [COLUMN] n] [TO [COLUMN] m | [LEN m]
 [DP {k | *}]]
 [DEFAULT 'value']
 [[READ] attributes] [REREAD attributes]
 [PRINT attributes]
```

```
TRANSFER [CONTROL] TO PROCESS {processname
```

```
 | %variable}
 [WITH] [USERID {%variable | 'string'}]
 [PASSWORD {variable | 'string'}]
 [ACCOUNT {%variable | 'string'}]
 [PASSING {IMAGE imagename | 'string' | %variable}]
```

```
UPDATE RECORD
```

```
VARIABLES ARE
 {FIXED [DP n] | FLOAT
 | STRING [LEN n] [DP {n | *}]
 | UNDEFINED}
```

```
[DECLARE] %variable [IS]
 {FIXED [DP n] | FLOAT} [ARRAY (d1[,d2[,d3]])]
 [COMMON]
```

```
[DECLARE] %variable [IS]
 STRING [LEN n] [DP {n | *}] [ARRAY (d1[,d2[d3]])]
[NO FIELD SAVE] [COMMON]
 %variable [(subscript)] = expression
```

```
+ WAIT [{n | %variable} SEC[S]] [FOR]
   {ANY RECEIPT RETURN %variable
   | RECEIPT {cid | processname | %variable}}
```

```
WRITE [IMAGE] imagename
 ON {sep-filename | TERMINAL | %variable}
```

Value specification syntax

For all User Language statements and retrieval conditions, wherever the term *value* appears in the syntax, it can be:

- Literal number or string
- VALUE IN clause of the form:

  ```
  VALUE [IN] label
  ```

- %variable

Retrieval condition syntax

A FIND statement can be followed by any number of retrieval conditions separated by an end of line or LINEND parameter character.

Conditions can be constructed as follows:

Syntax

```
[NOT] phrase [{AND | OR | NOR} [NOT] phrase] …
```

Where

phrase can be constructed as follows:

```
fieldname = [NOT] value

fieldname LIKE pattern

fieldname IS [NOT] {PRESENT | LIKE ’pattern’}

fieldname IS [NOT]

 {[NUMERICALLY | ALPHABETICALLY]

 [EQ | = | NE | ¬= | GREATER THAN | GT | >

 | LESS THAN | LT < | <= | GE | >= | BEFORE | AFTER]

 value}

fieldname IS [NOT]

 {[NUMERICALLY | ALPHABETICALLY]

 {IN RANGE [FROM | AFTER] value1 {TO | [AND] BEFORE}

 value2 | BETWEEN value1 AND value2}

FILES filename

FIND$ label

LISTS$ listname

LOCATION {location | =}

POINT$ value
SFGE$ value
SFL$ value
```

where *value* can be:

Literal number or string

`VALUE [IN] label`

%*variable*

Omitting repeated first words

Moreover, if a sequence of phrases in a particular retrieval condition all have the same first word, that word can be omitted from the latter phrases. For example:

```
LIST$ A AND NOT LIST$ B
```

can be written:

```
LIST$ A AND NOT B
```

And:

```
X IS 13 OR X IS LESS THAN 7
```

can be written:

```
X IS 13 OR IS LESS THAN 7
```

Omitting duplicated equal signs

Duplicated equal signs can be omitted. For example, the expression:

```
A = 3 OR A = 5 OR A = 40
```

is equivalent to:

```
A = 3 OR 5 OR 40
```

Use of parentheses

Parentheses can be placed around any sequence of phrases to clarify the condition or force the evaluation to occur in a particular order. For example:

```
NOT (A = 2 OR LIST$ Y)
A = 1 AND (B = 2 OR C = 3)
```

Print specification syntax

A PRINT, SET or AUDIT statement contains print specifications of the following form:

Syntax

```
[term] {AND | TAB | WITH]…[[term]
 [AND | TAB | WITH] …] … […]
```

Where

term can be constructed as follows:

```
{'string' | %variable | COUNT IN label
 | OCCURRENCE IN label
 | VALUE IN label | function}
 [AT [COLUMN] m] [TO [COLUMN] n]
```

or, if the statement is within a record loop:

```
{{EACH | n} fieldname | *RECORD | *ID}
 [AT [COLUMN] m] [TO [COLUMN] n}
```

Expression syntax

The following syntax can be used in:

- Assignment statements
- Conditional IF statements and ELSEIF clauses
- Computed JUMP TO statements
- Subscripts
- Function arguments

Syntax

```
{operand |(expression)} [operator {operand
| (expression)}] …
```

Where

operand can be constructed as follows:

```
[+ | - | NOT]
 {'string' | %variable | number
 | fieldname [(subscript)] IS [NOT] PRESENT
 | COUNT IN label | OCCURRENCE IN label
 | VALUE IN label
 | function}
```

and *operator* may be one of the following:

+	=	EQ	AND
-	¬=	NE	OR
*	>	GT	WITH
/	<	LT	IS PRESENT
	>=	GE	IS NOT PRESENT
	<=	LE	IS LIKE
			IS NOT LIKE

IN clause syntax

Statements preceded by an asterisk (*), beginning with “User Language statements” on page 26-4, support the IN clause.

The three basic forms of the IN clause are:

Syntax

-
  ```
  IN [PERMANENT | TEMPORARY ] GROUP groupname
  MEMBER [%member | [filename [AT {location | =}]]
  ```

-
  ```
  IN file1 [,file2] •••
  ```

-
  ```
  IN {$CURFILE | $UPDATE}
  ```
  The form IN $CURFILE can be used only within a record loop.

IN GROUP MEMBER limitations

In addition to the three basic forms of the IN clause shown above the IN GROUP MEMBER clause restricts the following statements to one member file in a group context:

- CLEAR LIST
- FIND ALL RECORDS (and its variants)
- FIND ALL VALUES
- FOR RECORD NUMBER (or FRN)
- STORE RECORD

You cannot use an IN GROUP MEMBER clause with a FOR EACH RECORD statement or with an ADD, CHANGE, or DELETE RECORD statement. Only the previously listed statements call accept an IN GROUP MEMBER clause.

Using an IN clause in a BEGIN…END block

The file name in the IN clause used within a BEGIN…END block is resolved by the compiler. You can either hard code a file name or use some type of dummy string for the file name. The use of a %variable for the file name is not allowed.

Subscript syntax

A subscript has the format:

Syntax `(subscript1 [,subscript2 [,subscript3]] )`

Where *subscript1*, *subscript2*, and *subscript3* can be any expression.

Terminal display attributes

One or more of the following terminal display attributes can replace the term *attribute* in a full-screen formatting statement, if the display attribute is supported by the installation:

List of attributes

BLINK

BLUE

BRIGHT

DIM

GREEN

INV[IS[BLE]]

NOBLINK

NOREV[ERSE]

NOU[NDER]SCORE

PINK

PROT[ECTED]

RED

REV[ERSE]

TURQUOISE

[UNDER]SCORE

UNPROT[ECTED]

VIS[IBLE]

WHITE

YELLOW

Type syntax for the DECLARE SUBROUTINE statement

The type specification for the DECLARE SUBROUTINE statement has one of the following formats:

```
{STRING [LEN n] [DP {n | *}] [ARRAY (*[,*[,*]])

 [NO FILE SAVE]]

 | FIXED [DP n] [ARRAY (*[,*[,*]])]

 | FLOAT}
```

or

```
[LIST] [IN [FILE | [PERM | TEMP] GROUP] name]
```

27

User Language Functions

In this chapter

- Standard functions
- Mathematical functions

Standard functions

Changes required to user-written $functions

In Version 5.1, $function operations were optimized to create CPU, QTBL and VTBL savings. Unfortunately, these optimizations break the open-coded setting of $function output using the RESULT macro.

The RESULT macro does not work in Model 204 Version 5.1. Change any use of the RESULT macro in $functions to use the standard LEAVENUM, LEAVEF0 and LEAVESTR macros.

$ACCOUNT

The $ACCOUNT function returns a variable-length character string equal to the account under which the user is logged into Model 204. If the login feature is not in use, $ACCOUNT returns the string ‘NO ACCOUNT'. This function takes no arguments. The ACCOUNT parameter returns the same value as the $ACCOUNT function. See the *Model 204 Command Reference Manual*.

Example

```
BEGIN
SET HEADER 1 ’MORTON CORPORATION’ -
   AT COLUMN 10
SET HEADER 2
SET HEADER 3 ’ACCOUNT: ’ AT COLUMN 10 -
   WITH $ACCOUNT
      .
      .
      .
```

$ACCT

$ACCT returns a variable-length character string equal to the user ID under which the user is logged into Model 204. If the login feature is not in use, $ACCT returns ‘NO ACCOUNT'. The $ACCT function takes no arguments.

$ALPHA

The $ALPHA function verifies whether a string is composed only of characters which are valid in the specified (or default) language. A 1 is returned if the condition is true; otherwise, a 0 is returned (for a false condition). A 0 is returned if there are any spaces or punctuation marks in the string, or if the string is null.

Syntax

The format of the $ALPHA function is:

```
$ALPHA(string [, language-name])
```

where:

- *string* represents the string to be verified. *string* must be one of:
  - Quoted literal
  - %variable
  - Unquoted field name, in which case the current value of the field is verified. In this case, the function call must be embedded in a FOR EACH RECORD loop.
- *language-name* (optional) specifies the language to use. Options are:
  - Omitting this argument, which instructs Model 204 to perform the validation for U.S. English, even if the value of the LANGUSER parameter is not NLANG.
  - A quoted asterisk (‘*'), which instructs Model 204 to use the value of the LANGUSER parameter to determine which language to use.
  - The quoted literal name of a valid language, for example: NLANGFR1 for French Canadian, Version 1. The request is cancelled with an error message, if the name is not present in NLANG$.

For example:

```
$ALPHA (’JOHN’) returns 1
$ALPHA (’JOHN SMITH’) returns 0
$ALPHA (’12A’) returns 0
```

Example

This request sorts and prints the names of agents whose name contains nonalphabetic characters. The quoted asterisk in the $ALPHA call causes Model 204 to verify the contents of the field AGENT against whatever language is indicated by the value of the LANGUSER parameter:

```
BEGIN
POL.HOLDERS:  FIND ALL RECORDS FOR WHICH
                 RECTYPE = POLICYHOLDER
              END FIND
              FOR EACH RECORD IN POL.HOLDERS
                 IF NOT $ALPHA (AGENT, ’*’) THEN
                    PLACE RECORD ON LIST BADNAME
                 END IF
              END FOR
ORDERED.LIST: SORT RECORDS ON LIST BADNAME BY AGENT
              FOR EACH RECORD IN ORDERED.LIST
                 PRINT AGENT
              END FOR
END
```

Note: For upward compatibility reasons, $ALPHA and $ALPHNUM do not recognize lowercase English letters as alphabetic characters unless a non-null language parameter is specified.

$ALPHNUM

The $ALPHNUM function verifies whether a string is composed only of characters which are valid in the specified (or default) language, and digits 0 through 9. A 1 is returned if the condition is true; otherwise, a 0 is returned (for a false condition). A 0 is returned if there are any spaces or punctuation marks in the string, or if the string is null.

Syntax

The format of the $ALPHNUM function is:

```
$ALPHNUM(string [, language-name])
```

where:

- *string* represents the string to be verified. *string* must be one of:
  - Quoted literal
  - %variable
  - Unquoted field name, in which case the current value of the field is verified. In this case, the function call must be embedded in a FOR EACH RECORD loop.
- *language-name* is optional and can be used to specify the language to use. Options are:
  - Omitting this argument, which instructs Model 204 to perform the validation U.S. English, even if the value of the LANGUSER parameter is not NLANG.
  - A quoted asterisk ('*'), which instructs Model 204 to use the value of the LANGUSER parameter to determine which language to use.
  - The quoted literal name of a valid language, for example: NLANGFR1, Version1, for French Canadian. The request is cancelled with an error message, if the name is not present in NLANG$.

For example:

```
$ALPHNUM ('JOHN') returns 1
$ALPHNUM ('JOHN SMITH') returns 0
$ALPHNUM ('12A') returns 1
```

Example

This request sorts by name and processes records whose designated field value does not meet the $ALPHNUM criteria. The second argument in the

$ALPHNUM call causes Model 204 to use U.S. English to perform the validation:

```
BEGIN
           %SEARCH =  $READ ('ENTER FIELD NAME')
FIND.RECS: FIND ALL RECORDS FOR WHICH
              RECTYPE = POLICYHOLDER
           END FIND
           PLACE RECORDS IN FIND.RECS ON LIST BAD
           FOR EACH RECORD IN FIND.RECS
              IF $ALPHNUM (%%SEARCH, 'NLANG') THEN
                 REMOVE RECORD FROM LIST BAD
              END IF
           END FOR
SORT.RECS: SORT RECORDS ON LIST BAD BY FULLNAME
           FOR EACH RECORD IN SORT.RECS
               .
               .
               .
END
```

Note: For upward compatibility reasons, $ALPHA and $ALPHNUM do not recognize lowercase English letters as alphabetic characters unless a non-null language parameter is specified.

$ARRSIZE

The $ARRSIZE function returns the number of elements in a particular dimension of a named array. This function is useful for users who pass entire arrays as parameters to a subroutine and then must know the size of the array supplied as the actual argument. For more information about passing arrays to a subroutine, refer to “Index loops” on page 11-19.

Syntax

The format of the $ARRSIZE function is:

$ARRSIZE (*name*, *dimension*)

where:

- *name* is a string that specifies the name of an array.
- *dimension* is a number that indicates the dimension of the named array for which the number of elements should be returned. Dimension can contain an expression whose value is 1, 2, or 3.

Example

```
FOR %I FROM 1 TO $ARRSIZE ('%COMM.ARRAY',1)
```

causes %I to iterate from 1 to the number of elements in dimension 1 of the array %COMM.ARRAY.

$ASCII

The $ASCII function converts an input string, assumed to be EBCDIC, into ASCII. For example:

```
%X = $ASCII(%X)
```

The translation table can be modified when necessary at customer sites. The source code is delivered in the FUNU module.

Not all strings are for display. There is no function to convert ASCII characters to EBCDIC characters. Question marks are usually the results of trying to print ASCII characters on an EBCDIC machine.

$BINARY

The $BINARY function converts the character string representation of a number into its equivalent fixed-point binary representation.

Syntax

The format of the $BINARY function is:

```
$BINARY (string [, precision [, scale]])
```

where:

- *string* is the character string to be converted.
- *precision* (optional) indicates the number of binary digits (bits) desired in the result of the function. Precision must be between 1 and 31. If the precision argument is greater than 15, a default value of 31 is used. If the precision argument is omitted, a default value of 15 (halfword) is used.
- *scale* (optional) indicates the number of fractional binary digits assumed in the result of the function. If the scale argument is omitted, a default value of 0 (integer) is used.

$BINARY returns a value of binary -1 if the string argument does not represent a valid number, if an invalid precision or scale value is specified, or if an overflow occurs.

$BLDPROC

The $BLDPROC function enables a request or series of requests to build a temporary procedure. $BLDPROC is similar to the PROCEDURE system control command.

The procedure built by $BLDPROC can contain arbitrary commands or User Language statements or other text. You can execute this procedure after the building request has ended, or you can edit the temporary procedure into a permanent procedure. For more information on temporary procedures, see “Working with temporary procedures” on page 13-20.

You can build only one procedure at a time. To add text to more than one procedure in rotation, you must close one procedure and reopen the next procedure.

Syntax

The format of the $BLDPROC function is:

```
$BLDPROC (proc number,text,action)
```

where:

- *proc number* is a temporary procedure number. The number of procedures or requests saved for a user is controlled by the NORQS parameter, which has a default value of 5. Procedure number 0 is the request currently being entered. Procedure -1 refers to the request entered before the most recent one, -2 to the one before that, and so on. Therefore, *proc number* must have a value between 0 and -NORQS+1.
- *text* is usually a single line to be appended to the temporary procedure. A single call to $BLDPROC can add more than one line of text by imbedding LINEND parameter characters (usually semicolons) in the text argument. If the text argument is null, the procedure is not changed.
- *action* must be one of the options listed in the following table. Building a temporary procedure is similar to building a sequential file in that the procedure must be opened before any text can be added to it. When all of the desired text has been added, the procedure should be closed.

  Choose from these options:

Option	Result
APPEND	Adds the text to the end of an already opened procedure.
CLOSE	Closes the temporary procedure, disallowing further APPENDs. Model 204 automatically closes any procedure left open at the end of execution of the request. Text specified in a CLOSE call is added before the close.
OPEN	Creates a new temporary procedure. If the procedure already exists, the old text is automatically deleted. Text specified in the same $BLDPROC call as OPEN is added after the procedure is opened.

Option	Result
REOPEN	Prepares an existing procedure for the addition of text. REOPEN locates the end of the old text so that new text is appended. OPEN and REOPEN are identical for a new procedure. Text specified in the same $BLDPROC call as REOPEN is added after the procedure is opened.

A null or omitted action argument is the same as APPEND.

How $BLDPROC works

$BLDPROC returns a 0 for success and a 1 for any of the following errors:

- The *proc number* argument is not numeric or is not in the range of valid temporary procedure numbers.
- The specified temporary procedure is being included.
- Where a previous USE PROC command is also directing output to the same temporary procedure.
- The *action* argument is not one of the valid choices.
- The *action* argument is OPEN or REOPEN and there is already an open temporary procedure.
- The *action* argument is APPEND or CLOSE and there is no open procedure.
- The action argument is APPEND or CLOSE and the proc number argument does not match the currently opened procedure.

Temporary procedures are stored in CCATEMP. If an additional CCAPTEMP page is required to process a $BLDPROC call, but CCATEMP is full, then the request is cancelled and the entire temporary procedure is deleted. After the request is cancelled, the procedure does not contain everything up to the point of failure. In the event of CCATEMP filling while processing $BLDPROC, the following message is issued:

```
*** CANCELLING REQUEST: M204.0441: CCATEMP FULL:
$BLDPROC
```

Example

This sample request saves the size of the global variable table (GTBL) before the table is reset with the UTABLE command. (See “User Language internal work areas” on page 21-3 for information on the UTABLE command and GTBL.) By saving the LGTBL value, the table can be returned to its original size at a later time. This is particularly useful in a subsystem where the LGTBL parameter is normally reset. (To learn about subsystems, see Chapter 23.)

```
BEGIN
```

```
%X = $BLDPROC(-1,'BEGIN','OPEN')
%X = $BLDPROC(-1,'%A = $SETG(''GTBL'',' WITH -
             $VIEW('LGTBL') WITH ')')
IF %A THEN
   PRINT 'GLOBAL TABLE FULL'
END IF
%X = $BLDPROC(-1,'END','CLOSE')
IF %X THEN
   PRINT 'BLDPROC ERROR'
END IF
END
UTABLE LGTBL 15000
```

After the preceding request executes, procedure -1 contains the following statements:

```
BEGIN
%A = $SETG('GTBL',704)
END
```

$CENQCT

The $CENQCT function reports the number of unused entries within the resource enqueuing table. The number must be more than the number of noncritical resources needed by the next statement(s) in the request. All resources except the following are noncritical:

```
DIRECT
EXISTENCE
INDEX
RECORD ENQUEUE
```

The function has no input and returns a single number. For example, 'PRINT $CENQCT' displays the number of unused entries.

$CHKMOD

Use the $CHKMOD function after a READ SCREEN or REREAD SCREEN statement to determine whether the terminal operator entered data for any full-screen input fields or for a specific input field. See a discussion of the full-screen feature in “Full-screen processing” on page 22-6.

Syntax

The format of the $CHKMOD function is:

```
$CHKMOD (screenname [, inputname])
```

- Both the *screenname* and *inputname* arguments are expected to be string expressions and can include quoted strings, %variables, field names, or functions.

If *screenname* and/or *inputname* are quoted strings, the name must be enclosed in single quotation marks:

```
$CHKMOD ('EMPSCRN', 'NAME')
```

- If *inputname* (representing an input field on the specified screen) is not included in the function call, $CHKMOD returns the number of input fields for which the terminal operator entered data.

  If *inputname* is specified, $CHKMOD returns a value of 0 if the input field was not modified. $CHKMOD returns a value of 1 if the field was modified.

$CHKPAT

The $CHKPAT function verifies the syntax of a pattern. $CHKPAT returns a null string if the pattern is syntactically correct, and an error message if it is not.

Without $CHKPAT, pattern syntax errors caused cancellation of the request or required the coding of sometimes awkward ON units.

$CHKPAT supports language-sensitive specification of patterns through an optional second parameter (such as “$ALPHA” on page 27-2 and “$ALPHNUM” on page 27-4).

All characters X’00’ through X’FF’ are valid in a pattern presented to the User Language pattern matcher. The $CHKPAT function no longer invokes either of the following messages when it encounters these characters.

```
M204.1688: errortype IN PATTERN 'pattern' AT
CHARACTER char
```

```
M204.1689: errortype IN PATTERN 'pattern' AT
CHARACTER char
```

Example

```
%X = CHKPAT(%PAT)
IF %X NE '' THEN
   PRINT %X
   JUMP TO ERROR.RETURN
END IF
```

$CHKPINF

Function

Returns information about checkpoints. It takes one argument, which specifies what information to return.

Syntax

```
$CHKPINF(request-code)
```

Where:

request-code	Asking to return…	$CHKPINF returns…	Meaning that checkpointing is…
0	Checkpoint status	0	Not active
		1	Currently being taken (by CPTIME or by CHECKPOINT command)
		2	Active, but no auto checkpoints; CPTIME=0
		3	Active and CPTIME does not equal 0
1	Date-time for the next checkpoint attempt, scheduled by the CPTIME argument	9999/99/99 99:99:99.99	Not active
		9999/99/99 99:99:99.99	CPTIME=0
		Current date-time	In progress
		yyyy/mm/dd hh:mm:ss.hh	Scheduled date/time
2	Seconds till the next checkpoint attempt, scheduled by the CPTIME parameter	999999999	Not active
		999999999	CPTIME=0
		0	In progress
		sss:hh	Scheduled in this many seconds
3	Time of last successful checkpoint	Date-time of last checkpoint as *yyyy/mm/dd hh:mm:ss.hh*	Due to CPTIME parameter or CHECKPOINT command
		0000/00/00 00:00:00.00	Not active
4	Seconds since the last successful checkpoint	Time in seconds of checkpoint as *sss:hh*	Due to CPTIME parameter or the CHECKPOINT command
		999999999	Not active
5	Total number of records currently in CHKPOINT stream, which includes checkpoints and preimages	Number of records	
6	Number of checkpoints currently in CHKPOINT stream	Number of checkpoints	
7	Extended quiesce status	0	Extended quiesce inactive for this run
		1	Extended quiesce unset: will not be entered

request-code	Asking to return…	$CHKPINF returns…	Meaning that checkpointing is…
		2	Extended quiesce set; at end of next successful checkpoint extended quiesce state will be reentered
		3	Currently in extended quiesce
		4	Extended quiesce facility non-functional: in EOJ

Example

The following procedure illustrates using the $CHKPINF function.

```
 * * * TOP OF PROCEDURE * * *
BEGIN
PRINT $CHKPINF(0) WITH ’ CURRENT CHECKPOINTING STATUS’ AT 25
PRINT $CHKPINF(1) WITH ’ NEXT SCHEDULED CHECKPOINT’ AT 25
PRINT $CHKPINF(2) WITH ’ SECONDS UNTIL NEXT SCHEDULED CHECKPOINT’
AT 25
PRINT $CHKPINF(3) WITH ’ LAST SUCCESSFUL CHECKPOINT TAKEN’ AT 25
PRINT $CHKPINF(4) WITH ’ SECONDS SINCE LAST SUCCESSFUL CHECK-
POINT’ AT 25
PRINT $CHKPINF(5) WITH ’ # RECORDS CURRENTLY IN CHKPOINT STREAM’
AT 25
PRINT $CHKPINF(6) WITH ’ # CHECKPOINTS IN CHKPOINT STREATM’ AT 25
PRINT $CHKPINF(7) WITH ’ EXTENDED QUIESCE STATUS’ AT 25
END
 * * * BOTTOM OF PROCEDURE * * *
```

Model 204 displays the following output:

```
3                       CURRENT CHECKPOINTING STATUS
2001/11/19 12:40:00.75  NEXT SCHEDULED CHECKPOINT
249.68                  SECONDS UNTIL NEXT SCHEDULED CHECKPOINT
2001/11/19 12:20:00.66  LAST SUCCESSFUL CHECKPOINT TAKEN
950.42                  SECONDS SINCE LAST SUCCESSFUL CHECKPOINT
7083                    # RECORDS CURRENTLY IN CHKPOINT STREAM
1                       # CHECKPOINTS IN CHKPOINT STREAM
1                       EXTENDED QUIESCE STATUS
```

$CHKSFLD

Use the $CHKSFLD function to determine whether a display attribute is applied to a screen item for a specified type of screen processing.

Syntax

```
$CHKSFLD(screen-name,item-name,display-attribute,
        [process-type])
```

Where

- Both the *screen-name* and *item-name* arguments are expected to be string expressions and can include quoted strings, %variables, field names, or functions.
- *display-attribute* value can be one of the following:

Value	Meaning
‘ASK'	Auto-skip
‘BLI'	Blink
‘BLU'	Blue color
‘BRI'	Bright
‘DEF'	Default color
‘GRE'	Green color
‘INV'	Invisible
‘NUM'	Numeric field
‘PIN'	Pink color
‘PRO'	Protected
‘RED'	Red color
‘REV'	Reverse image
‘TUR'	Turquoise color
‘USC'	Underscore
‘WHI'	White color
‘YEL'	Yellow color

- *process-type* can be one of the following:
  - ‘PRINT'
  - ‘READ' (the default)
  - ‘REREAD'
  - ‘TAG'

Returns

- 1 if the attribute is ON for the type specified.
- 0 if the attribute is OFF for the type specified.

Usage

For example, if you want to check whether the POLNO (policy number) field in the screen MAIN is displayed in blue in reread processing, enter the following command:

```
BEGIN
SCREEN MAIN
INPUT POLNO AT 10 LEN 20 RED
END SCREEN
MODIFY %MAIN:POLNO TO BLUE FOR REREAD
PRINT $CHKSFLD('MAIN','POLNO','BLU','REREAD')
END
```

Produces the following output:

```
1
```

Nonfatal errors

If you do not specify colors within the Model 204 screen definition, the color displayed on your terminal is the default color display. For example, you might be looking at a pink display, but when you invoke the $CHKSFLD function, checking for ‘PIN' does not return a ‘1' because pink was not assigned by the Model 204 screen definition.

The following messages might be issued:

```
M204.2462: INVALID ATTRIBUTE SPECIFIED: attribute

M204.2462: INVALID ATTR.TYPE SPECIFIED: attr.TYPE

M204.0329: SCREEN OR IMAGE ITEM NAME NOT FOUND: name

M204.0324: SCREEN, MENU, OR IMAGE NAME NOT DEFINED:
name

M204.0247: SCREEN, MENU, OR IMAGE NAME NOT SPECIFIED
```

$CHKTAG

Use the $CHKTAG function to determine whether any full-screen input fields that resulted in error conditions were entered by the terminal operator. The full-screen input fields either did not pass the automatic validation tests specified for the fields, or contained errors detected by the request.

Syntax

The format of the $CHKTAG function is:

```
$CHKTAG (screenname [, inputname])
```

- Both the *screenname* and *inputname* arguments are expected to be string expressions and can include quoted strings, %variables, field names, or functions.

  If *screenname* and/or *inputname* are quoted strings, the name must be enclosed in single quotation marks, as shown below:

  ```
  $CHKTAG (%SCRNAME, ’ACCNTNO’)
  ```

- If *inputname* (representing an input field on the specified screen) is not included in the function call, $CHKTAG returns the number of input items with tags on. If inputname is included, $CHKTAG returns a value of 0 if the specified input item’s tag is not on. $CHKTAG returns a value of 1 if the item’s tag is on.

Example

```
IF $CHKTAG(’ACCTG’) GT 0
   THEN REREAD SCREEN ACCTG
```

$CODE

The $CODE function, along with $DECODE, provides an encoding/decoding facility in User Language. $CODE and $DECODE operate on code tables that are defined, created, and controlled by the system manager. These tables can be searched but not changed by User Language functions.

$CODE and $DECODE search tables that are external to a Model 204 file. These tables are independent of a field’s CODED/NON-CODED attribute, which affects how the field is stored internally in the Model 204 file.

The $CODE function takes two arguments:

- The first argument contains the name of the table to be searched.
- $CODE returns the code for the character string value contained in the second argument. A null value is returned if the string is not contained within the table or if the table does not exist. A nonexistent table also causes the nonfatal message:

  ```
  INVALID CODE TABLE IDENTIFIER=X.
  ```

Example

This request prompts the user for the full name of a state but performs the record search on the coded value for that state.

```
BEGIN
       %STCD = $CODE ('STATE', $READ ('ENTER STATE
NAME'))
GET.RECS: FIND ALL RECORDS FOR WHICH
             STATE = %STCD
          END FIND
          FOR EACH RECORD IN GET.RECS
             .
             .
             .
```

$CURFILE

The $CURFILE function returns the name of the file from which the current record has been selected. If the file is remote, $CURFILE also returns the location of the file (in the form *`filename`* `AT` *`location`*). $CURFILE takes no arguments.

You can use $CURFILE in two places:

- In arithmetic and PRINT specifications within a record loop
- In an IN clause to override a default file or group in a STORE RECORD statement

See "$CURFILE and $UPDATE functions" on page 16-29 for a discussion of default files and for examples of requests using $CURFILE.

You cannot use an IN clause that includes both MEMBER and $CURFILE. See "IN GROUP MEMBER clause" on page 16-27 for more information about the IN GROUP MEMBER clause.

$CURREC

The $CURREC function returns an integer equal to the internal number of the current record.

At the beginning of a request, $CURREC is set to minus one, an invalid record number. The STORE RECORD statement sets $CURREC to the record number of the new record.

At the beginning of each pass through a FOR loop, $CURREC is set to the record number of the record to be processed in that pass. When the FOR loop is exited, because all records have been processed or because a LOOP END or JUMP statement has been encountered, $CURREC is restored to its value prior to the FOR statement. $CURREC takes no arguments.

You should be aware of these facts when using $CURREC:

- In a record loop on the records of a SORT statement, $CURREC is set to the record number of the record from which the temporary sort copy was made.

- Record numbers are not unique within a file group. The number returned by $CURREC is valid only in reference to the file from which the record came.

Example

This request prints the record number of a new record:

```
BEGIN
GET.RECS: FIND ALL RECORDS
          END FIND
          FOR 5 RECORDS IN GET.RECS
             PRINT 'THE INTERNAL RECORD NO IS ' -
                WITH $CURREC
          END FOR
END
```

$C2X

The $C2X function translates each byte within a character string into two-byte hexadecimal-equivalent characters. $C2X returns a character string that is twice as long as the original string. The maximum input length is 126 bytes. If the input length is more than 126 bytes, a null string is returned.

Syntax

The format for the $C2X function is:

`$C2X(`*`charstr`*`)`

where *charstr* is the input character string (either a %variable or a quoted literal)

Example

```
PRINT $C2X('YES')
```

results in this output:

```
E8C5E2
```

Overview of $DATE functions

All $DATE functions—$DATE, $DATEJ, $DATEP—accept two input arguments.

Syntax `$DATE(`*`year-format,fill-character`*`)`

Where

- The *year-format* argument controls the format of the year based on the following input values:

Year-format input	**Return format**	**Example: Year 2013 displayed as…**
0	YY	13
1	YYYY	2013
2	CYY	113 (The first digit represents the century since 1900)

CYY represents the century-year format. The first digit represents the century since 1900. The CYY format can be manipulated using the CUSTOM parameter. Please consult the *Model 204 Command Reference Manual* for a discussion of the CUSTOM parameter.

- The *fill-character* argument indicates a 1-byte fill character to place between the date components, as in the following examples:.

Code example	**Return format**	**Example: July 11, 2013**
$DATE(2,' ')	'CYY MM DD'	113 07 11
$DATE(0,'Z')	'YYzMMzDD'	13z07z11
$DATEJ(1,'-')	'YYYY-DDD'	2013-192
$DATEJ(1,'')	'YYYYDDD'	2013192

Usage

$DATECHG, $DATECHK, $DATECNV, and $DATEDIF recognize a format of CYY as representing the century-year format as returned from the $DATE function. Conversion to and from the CYY format is fully supported.

In addition, if CUSTOM=1 is added to the User 0 CCAIN stream or set later using the RESET command, the following occurs: If a CYY date format is specified for conversion and only a YY input is supplied, the conversion is successfully completed by using the CENTSPLT and BASECENT parameters. For further details see *Model 204 Command Reference Manual* for details regarding the BASECENT, CENTSPLT, and CUSTOM parameters.

Examples

- IF CUSTOM=1, BASECENT=19, and CENTSPLT=95, then:

Print command	**Results**
`PRINT $DATECNV('CYYDDD','YYYYMMDD','96001')`	`19960101`
`PRINT $DATECNV('CYYDDD','YYYYMMDD','196001')`	`20960101`
`PRINT $DATECNV('CYYDDD','YYYYMMDD','95001')`	`19950101`

Print command	Results
`PRINT $DATECNV('CYYDDD','YYYYMMDD','095001')`	`19950101`

- If CUSTOM=1 is omitted, BASECENT=19, and CENTSPLT=95, then:

Print command	Results
`PRINT $DATECNV('CYYDDD','YYYYMMDD','96001')`	`******`
`PRINT $DATECNV('CYYDDD','YYYYMMDD','196001')`	`20960101`
`PRINT $DATECNV('CYYDDD','YYYYMMDD','95001')`	`******`
`PRINT $DATECNV('CYYDDD','YYYYMMDD','095001')`	`19950101`

- If CUSTOM=2 is added to the User 0 CCAIN stream or set later using the RESET command, the following occurs: If a CYY-format is specified for output conversion and the C indicator is zero, then C is eliminated. This is true for all $DATE function calls. For example:

  If CUSTOM=2 then: PRINT $DATEJ(2) prints 97.001.

  But if CUSTOM =2 is omitted, then PRINT $DATEJ(2) prints 097.001.

Julian dates

The Julian date is a 4-byte, packed decimal formatted as follows:

```
0CYYDDDF
```

where:

- 0 is a reserved filler
- *C* represents the century since 1900. For example:
  - C=0 represents years 1900-1999
  - C=1 represents years 2000-2099
- *YY* is 2-byte year
- *DDD* is 3-byte day
- *F* is positive sign nibble for packed decimal

Routines available for user-written date $functions

CCALL entry points have been added: DATE, DATE3, and DATE4. All routines must be called with T1 pointing to a 26-byte answer area. CCA recommends that you allocate the answer area using the VARS=(name, len) pushdown list variable of the ENTER macro. (See the *Model 204 System Manager's Guide* chapter on customizing function and translation tables for more information on the ENTER macro.)

The current date and time are stored in the area with this format:

Code entry point	Format used for storing…
`DATE`	`'YY.DDD MON DD HH.MM.SS'`
`DATE3`	`'CYY.DDD MON DD HH.MM.SS'`
`DATE4`	`'YYYY.DDD MON DD HH.MM.SS'`

All registers are returned intact with the exception of the DATE call. The DATE call changes only the T4 register, returning the number representing the current month (1-12).

$DATE

The $DATE function returns an 8- or 10-character string equal to the current date in yy-mm-dd format (for example, 90-09-20) or yyyy-mm-dd format (for example, 1990-09-20). The default is 8 characters. If $DATE is stored as a field value, you can use this form to sort records chronologically.

Usage

When using $DATE in field values, make sure that all values of $DATE are in the same format. You will get incorrect results when sorting records if you mix yyyy and yy formats.

Syntax

The format for $DATE is:

`$DATE(`*`year-format,fill-character`*`)`

See "Overview of $DATE functions" on page 27-17 for syntax details.

If Model 204 encounters an error, the function returns all asterisks (*).

Example

The following request prompts a user for data values and automatically stores the current date with each new record (using the 8-character format).

```
BEGIN
%DATE = $DATE
GET.FIRST: %A = $READ('ENTER FIELD A')
           %B = $READ 'ENTER FIELD B')
           STORE RECORD
              FLD A = %A
```

```
                FLD B = %B
                   .
                   .
                   .
                DATE STORED = %DATE
             END STORE
             IF $READ('NEXT RECORD: Y OR CR') EQ -
                'Y' THEN JUMP TO GET.FIRST
             END IF
END
```

$DATECHG

The $DATECHG function adds or subtracts a specified number of days from a given date. The result is returned in the format of the input date.

Syntax

The format of the $DATECHG function is:

`$DATECHG(`*`format, date, number of days`*`)`

where:

- *format* specifies the format of the input date. The format can be a combination of these elements:

DD	Gregorian numeric day
DDD	Julian numeric date
MM	Numeric month
MON	Abbreviated month name
MONTH	Full month name
YY	Last two digits of numeric year (assumes that the year prefix is 19)
YYYY	Full numeric year
CYY	The century, plus the year. Century (C) is a single digit, where 0 represents 1900, 1 represents 2000, and so on.

  Valid formats are:

  – A format that has a month, day, and year element

  – A Julian date format that has a year element and a day element in the format DDD

  Any EBCDIC characters except single quotes are allowed within the format. The format can be as many as 32 characters in length.

- *date* specifies a date in the format indicated by the format argument. The date can be as many as 36 characters in length.

- *number of days* indicates the number of days to be added or deducted from the date. The number must be an integer.

Separators and leading zeros

- The separators in the format argument must match the separators in the date argument. For example:

  ```
  $DATECHG(’MONTH - DD - YYYY’,’JANUARY - 05 -
  1986’, 20)
  ```

- When necessary, you must pad the month or date in the date argument with leading zeros to match the length of the format. For example:

  ```
  $DATECHG(’YY DDD’,’86 023’,22)
  ```

How $DATECHG works

- If the number of days is a positive integer, the number is added to the date.
- If number of days is a negative integer, the number is deducted from the date.
- If an error occurs, all asterisks (*) are returned.

Example

```
PRINT $DATECHG(’MON. DD, YYYY’,’JAN. 10, 1999’, 15)
```

prints this value:

```
JAN. 25, 1999
```

$DATECHK

The $DATECHK function determines whether a given date is valid. $DATECHK returns a 1 if the date is valid. A 0 is returned if either the date is invalid or if the date does not match a format you specify.

Syntax

The format of the $DATECHK function is:

$DATECHK(*format, date*)

where:

- *format* specifies the format of the date, which can be a combination of the following elements:

DD	Gregorian numeric day

DDD	Julian numeric date
MM	Numeric month
MON	Abbreviated month name
MONTH	Full month name
YY	Last two digits of numeric year (assumes that the year prefix is 19)
YYYY	Full numeric year
CYY	The century, plus the year. Century (C) is a single digit, where 0 represents 1900, 1 represents 2000, and so on.

Valid formats are:

– A format that has a month, day, and year element

– A Julian date format that has a year element and a day element in the format DDD

Any EBCDIC characters except single quotes are allowed within the format. The format can be as many as 32 characters in length.

- *date* specifies a date in the format indicated by the format argument. The date can be as many as 36 characters in length.

Example

```
PRINT $DATECHK('MON. DD, YYYY','FEB. 30, 1999')
```

prints the value 0 because the month of February does not have 30 days.

$DATECNV

The $DATECNV function converts an input date from its current format to a format you specify and, also, determines whether the input date is valid. If a format error occurs or the input date is not valid, the function returns all asterisks (*). $DATECNV supports both a 2- and a 4-digit year format; the year prefix can come from one of four places.

Syntax

The format of the $DATECNV function is:

```
$DATECNV(input format,output format,input date,
          defcent, centsplt)
```

where:

- *input format* specifies the format of the input date. The format can be a combination of these elements:

DD	Gregorian numeric day
DDD	Julian numeric date
MM	Numeric month
MON	Abbreviated month name
MONTH	Full month name
YY	Last two digits of numeric year (assumes that the year prefix is 19)
YYYY	Full numeric year
CYY	The century, plus the year. Century (C) is a single digit, where 0 represents 1900, 1 represents 2000, and so on.

  Valid formats are:

  – A format that has a month, day, and year element

  – A Julian date format that has a year element and a day element in the format DDD

  Any EBCDIC characters except single quotes are allowed within the input format and appear unchanged in the output date. The input format can be as many as 32 characters in length.

  When you use a 2-digit year, for example, YY is 98) in the input format and a 4-digit year (YYYY) in the output format, Model 204 assumes that the century is 19-, for example, 1998.

- *output format* specifies the format of the output date. The format requirements are the same as those for the input format argument. Using a 2-digit year (YY) in the output format leaves you with no way to distinguish between centuries. For example:

```
$DATECNV('MON DD, YYYY', 'YY DDD', 'JAN 10, 2005')
```

  produces the following output:

```
05 010
```

  However, the output is exactly the same if the input date is January 10, 1905 or January 10, 2105.

- *input date* specifies the input date in the format indicated by the *input format* argument. The input date can be as many as 36 characters in length.

- *defcent* argument (optional) specifies the DEFCENT value to use; it overrides all other DEFCENT and CENTSPLT parameter values. This argument cannot be specified with the *centsplt* argument, unless one of the values is NULL.

- *centsplt* argument (optional) specifies the CENTSPLT value to use; it overrides all other DEFCENT and CENTSPLT parameter values. This argument cannot be specified with the *defcent* argument, unless one of the values is NULL.

The following example illustrates the use of the $DATECNV converting dates in various centuries.

```
PROCEDURE CENTSPLT
BEGIN
%INPUT_FORMAT = ’YYDDD’
%OUTPUT_FORMAT = ’YYYY MM DD’
%INPUT_VALUE = ’9633’
%CENTSPLT = 97
%DEFCENT = ’ ’
CALL PIT
%CENTSPLT = ’ ’
%DEFCENT = 19
CALL PIT
%INPUT_FORMAT = ’YYYYDDD’
%INPUT_VALUE = ’2006333’
CALL PIT
PIT: SUBROUTINE
PRINT %INPUT_FORMAT ’ ’ %OUTPUT_FORMAT ’ ’
%INPUT_VALUE -
       ’ ’ %DEFCENT ’ ’ %CENTSPLT
PRINT $DATECNV(%INPUT_FORMAT,
%OUTPUT_FORMAT,%INPUT_VALUE, -
       %DEFCENT, %CENTSPLT)
RETURN
END SUBROUTINE
END
END PROCEDURE
```

For an explanation of the DEFCENT and CENTSPLT parameters processing see *Model 204 Command Reference Manual* in the chapter on parameters.

Separators and leading zeros

Use separators and leading zeros as specified here:

- Separators in the *input format* argument must match the separators in the *input date* argument. For example:

```
$DATECNV(’MONTH - DD - YYYY’,’MON DD, YYYY’, -
      ’JANUARY - 05 - 1990’)
```

- When necessary, you must pad the month or date in the *input date* argument with leading zeros to match the length of *input format*. For example:

```
$DATECNV(’YY DDD’,’MON DD, YYYY’,’90 023’)
```

Example

```
PRINT $DATECNV('DDMMYY','MON DD, YYYY','010790')
```

prints this value:

```
JUL 01, 1990
```

$DATEDIF

The $DATEDIF function returns the difference in days between the two dates.

Syntax

The format of the $DATEDIF function is:

```
$DATEDIF date-1-format,date-1,date-2-format,date-2,
          defcent,centsplt)
```

where:

- *Date-1-format* specifies the format of the first date. The format can be a combination of these elements:

DD	Gregorian numeric day
DDD	Julian numeric date
MM	Numeric month
MON	Abbreviated month name
MONTH	Full month name
YY	Last two digits of numeric year (assumes that the year prefix is 19)
YYYY	Full numeric year
CYY	The century, plus the year. Century (C) is a single digit, where 0 represents 1900, 1 represents 2000, and so on.

  Valid formats are:

  – A format that has a month, day, and year element

  – A Julian date format that has a year element and a day element in the format DDD

  Any EBCDIC characters except single quotes are allowed within the input format and appear unchanged in the output date. The input format can be as many as 32 characters in length.

- *Date-1* specifies the first date in the format specified by the *date-1-format* argument. The date can be as many as 36 characters in length.

- *Date-2-format* specifies the format of the second date. The format requirements are the same as those for the *date-1-format* argument. This argument can be omitted but the comma after it is required. If this argument is omitted, $DATEDIF assumes that *date-2* is in the same format as *date-1*.
- *Date-2* specifies the second date. The date must be either in the format specified in the *date-2-format* argument, or in the format specified in *date-1-format* if *date-2-format* is not specified. The date can be as many as 36 characters in length.
- The *defcent* argument (optional) specifies the DEFCENT value to use; it overrides all other DEFCENT and CENTSPLT parameter values. This argument cannot be specified with the *centsplt* argument, unless one of the values is NULL.
- The *centsplt* argument (optional) specifies the CENTSPLIT value to use; it overrides all other DEFCENT and CENTSPLT parameter values. This argument cannot be specified with the *defcent* argument, unless one of the values is NULL.

Dates in differing centuries

In the following procedure Date_Difference, the $DATEDIF function calculates the difference between dates twice. Before the first PRINT command, the CUSTOM parameter is set to 1, so the century defaults to the current century. Before the second PRINT command, the DEFCENT parameter is set to 20.

```
PROCEDURE DATE_DIFFERENCE
RESET CUSTOM =(1

BEGIN
PRINT $DATEDIF(’CYYDDD’,’96333’,’YY MM DD’,’97 06 22’,-
      %DEFCENT,%CENTSPLT)
%DEFCENT = 20
PRINT $DATEDIF(’CYYDDD’,’096333’,’YY MM DD’,’97 06 22’,-
      %DEFCENT,%CENTSPLT)
END
```

```
END PROCEDURE
```

For an explanation of the DEFCENT and CENTSPLT parameters processing see *Model 204 Command Reference Manual* in the chapter on parameters.

Separators and leading zeros

- The separators in each format must match the separators in the corresponding date. For example:

  ```
  $DATEDIF(’MON DD, YYYY’,’JAN 08, 1990’, -
          ’YY:DDD’, ’88:210’)
  ```

- When necessary, pad the month or date in the date argument with leading zeros to match the length of corresponding format argument. For example:

```
   $DATEDIF('YY DDD','90 034',,'90 007')
```

How $DATEDIF works

- If date1 is the same as date2, 0 is returned.
- If date1 is earlier than date2, a negative integer is returned.
- If date1 is later than date2, a positive integer is returned.
- If an error occurs, 999999999 is returned.

Example

```
PRINT $DATEDIF('MMDDYY','010790',,'040891')
```

prints this value:

```
-456
```

$DATEJ

The $DATEJ function returns the current Julian date as a 5- to 7-character string in yy-ddd, format, for example, 97-342; or cyy-ddd format, for example, 097-342; or yyyy-ddd format, for example, 1997-342. The default is 5 characters. If $DATEJ is stored as a field value, you can use this form to sort records chronologically.

Usage

When using $DATEJ in field values, make sure that all values of $DATEJ are in the same format. You get incorrect results when sorting records if you mix yyyy, cyy, and yy formats.

Syntax

The format for $DATEJ is:

```
$DATEJ(year-format,fill-character)
```

For further syntax details see “Overview of $DATE functions” on page 27-17.

If Model 204 encounters an error, the function returns all asterisks (*).

$DATEP

The $DATEP function returns a 9-character or 11-character string equal to the current date in either “dd mon yy” format (for example, 04 AUG 91) or “dd mon yyyy” format (for example, 04 AUG 1991). The default is 9 characters.

Syntax

The format for $DATEP is:

```
$DATEP[(year-format)]
```

where *year-format* is optional and can be:

0	The default; indicates that $DATEP returns a 9-character string equal to the current date in the dd mon yy format.
1	Indicates that $DATEP returns an 11-character string equal to the current date in the dd mon yyyy format.

If Model 204 encounters an error, the function returns all asterisks (*).

See also “Overview of $DATE functions” on page 27-17.

Example

```
BEGIN
SET HEADER 1 ’TRIAL BALANCE’ WITH $DATEP AT COLUMN 30
NEW PAGE
   .
   .
   .
```

$DAY

The $DAY function takes an input day number and returns a 3-byte string containing the name of the day in ascending order beginning with Sunday. The return values are: SUN, MON, TUE, WED, THR, FRI, and SAT.

Syntax

```
$DAY(day-number)
```

Where:

Day-number is the number for the day of the week

Usage

For example, if you enter the following command:

```
PRINT $DAY(5)
```

The Model 204 displays:

```
’THR’
```

If CUSTOM=3 is selected, $DAY function takes an input day number from 2 through 8 representing in ascending order, Monday through Sunday. A string containing the full name of the day is returned, for example, if you enter the following command:

```
PRINT $DAY(5)
```

The Model 204 displays:

```
’FRIDAY’
```

Also, 9 represents the string ‘MON-FRI'.

$DAYI

The $DAYI function takes an input day number and the century-split and resolves the date. The value returned is a number reflecting the day of the week for the given date.

Syntax

```
$DAYI(input-date,century-split)
```

Where:

- *input-date* is a string in the format CYYDDD. (C is the century; YY is the year, and DDD is the julian day.)
- *century-split* is the 2-digit number representing the lowest year in the current century. (Valid values are 00 to 99.)

Usage

If the century (C) value is omitted, the century is determined by one of the following

- The supplied century split value
- The system defined century split value (CENTSPLT)
- The system default century (DEFCENT)

If the year is omitted, the current Model 204 defined year is used. The Julian day is required.

The default operation is CUSTOM=0, where the week days are numbered from 1 through 7, representing Sunday through Saturday. If CUSTOM=3 is selected, the week consists of 2 through 8, representing Monday through Sunday.

Examples

If the default century is 1900, then:

Table 27-1. Resolving $DAYI function

Input-date	Representing	Returns	Representing
$DAYI(‘100032')	February 1, 2000	3	Tuesday
$DAYI(97001',500)	January 1, 1997	4	Wednesday
$DAYI(‘23001',50)	January 1, 2023	2	Sunday

$DEBLANK

The $DEBLANK function is equivalent to the $SUBSTR function, except that the resulting string is stripped of leading and trailing blanks. Refer to the discussion on $SUBSTR on page 27-103 for more information.

$DECODE

The $DECODE function performs a table lookup and returns a decoded character string value. $DECODE is useful for printing descriptive information on reports when the records themselves contain only coded values.

$DECODE conserves space and minimizes the keystrokes required for requests by retaining only codes for fields. $DECODE takes two arguments:

```
$DECODE('table_name',resolve)
```

where

- *table_name* is the name of a table.
- *resolve* contains the code that is to be decoded.

Example

The state field contains a 2-character code. Full state names are printed.

```
BEGIN
STATE:      FOR EACH VALUE OF STATE IN ORDER
STATE.VAL:     FIND ALL RECORDS FOR WHICH
                  STATE = VALUE IN STATE
               END FIND
NO.IN.EACH:    COUNT RECORDS IN STATE.VAL
               SKIP 1 LINE
               PRINT $DECODE ('STATE', VALUE IN STATE) -
                  WITH COUNT IN NO.IN.EACH AT COLUMN 30
            END FOR
END
```

$DELG

Deletes information stored in the global variable table by a $SETG function in the same or an earlier request.

The syntax of the $DELG function is:

```
$DELG('globalvariable[*]')
```

Where:

globalvariable[*] specifies the name of the global variable to delete.

You can specify the optional wildcard suffix, an asterisk (*), to delete all global variables with the same common prefix. You cannot delete all global variables by specifying the wildcard suffix only.

$DELG returns a completion code indication success or failure of the operation. Possible codes are:

Code	Meaning
0	Successful. Global variable(s) deleted.
1	Not found. No global variable found that matches the supplied argument.
2	Unsuccessful.

Examples

Assuming the global variable table has been populated as follows:

```
%RC = $SETG('GVAR','DATA')
%RC = $SETG('GVAR1','DATA1')
%RC = $SETG('DELETE','TARGET')
%RC = $SETG('GVAR2','DATA2')
%RC = $SETG('REMAINING','ENTRY')
%RC = $SETG('GVAR3','DATA3')
```

Then the following $DELG functions return:

Function	Returns…	And…
`%RC=$DELG('')`	1	Does not alter the global variable table
`%RC=$DELG('DELETE')`	0	Removes the entry DELETE from the global variable table
`%RC=$DELG('NOTFOUND')`	1	Does not alter the variable table
`%RC=$DELG('GVAR*')`	0	Removes all GVAR*n* entries, including GVAR, from the global variable table

Function	Returns…	And…
`%RC=$DELG('*')`	1	Does not alter the global variable table

The resulting global variable table is:

```
REMAINING=ENTRY
```

For a full discussion of global variables refer to Chapter 20.

$DSCR

The functionality of the $DSCR function is superseded by the $LSTFLD and $FDEF functions. $DSCR is still supported, however. See Appendix A for detailed information on this function.

$DSN

$DSN is useful with Model 204 files that are comprised of multiple datasets. You specify the Model 204 file's DD name and the ordinal number (first, second, third, and so on) of one of the file's datasets, and $DSN returns the dataset's name.

The $DSN syntax is:

```
$DSN('ddname',dsnnum)
```

where:

- *ddname* is a Model 204 file's DD or file name.
- *dsnnum* is a positive integer that represents the ordinal number of a dataset. For example, specify 3 for *dsnnum* if you want the name of the third of the multiple datasets that comprise the Model 204 file.

If no file or DD name is specified, if the dataset number is too large, and if Model 204 cannot find the dataset name, the system returns a counting error message.

Examples

An example using $DSN follows:

```
FOR %I FROM 1 TO $DSNNUM('FU',%I) BY 1
  PRINT $DSN('FU',%I)
END FOR
```

The next example does a straight $DSN call for a file mapped to one dataset.

```
%FILENAME IS STRING LEN 8
%FILENAME = 'VEHICLES'
PRINT 'FILE ' WITH %FILENAME WITH ' IS CONTAINED IN -
```

```
DATASET ’ WITH $DSN(%FILENAME,1)
```

$DSNNUM

$DSNNUM is useful with Model 204 files that are comprised of multiple datasets. You specify the Model 204 file’s DD name or file name and $DSNNUM returns the total number of datasets defined for the file.

The $DSNNUM syntax is:

$DSNNUM(’*ddname*’)

where:

ddname is a file or dataset’s DD name.

$DSNNUM returns a zero if the file or dataset is not found. It returns a -1 for an argument syntax error.

For a sequential file, the number of dataset names is always one.

The following example combines the use of $DSNNUM to identify the number of datasets that a file maps to, and then uses $DSN to print out the dataset name.

```
%FILENAME IS STRING LEN 8
%FILENAME = ’MYFILE’
PRINT ’FILE ’ WITH %FILENAME WITH -
 ’IS CONTAINED IN THE FOLLOWING DSNS ’ -
PRINT
FOR %I FROM 1 TO $DSNNUM(%FILENAME)
 PRINT $DSN(%FILENAME,%1)
END
```

$ECBDGET

Function

Get string data associated with an Event Control Block (ECB)

Syntax

```
$ECBDGET(ECB-number | ’CPQZ’)
```

Where:

- *ECB-number* is a string with a numeric value from one to the NECBS parameter that identifies the ECB from which to retrieve the text. The *ECB-number* can be expressed as a numeric literal, a %variable, or a field name.

- CPQZ is a named ECB used by the NonStop/204 facility to automatically post an extended quiesce. See the *Model 204 System Manager's Guide* for an explanation of the facility.

  To use CPQZ, you need not set the NECBS parameter. CPQZ can be expressed as a literal, a %variable, or a field name. When CPQZ is specified, the value of the $ECBDGET function can be non-null (except for error values) during only extended quiesce. CPQZ is internally cleared to null when the system exits extended quiesce.

Usage

When the $ECBDGET function is successful, it returns your data as a text string up to 255 bytes long. If you make a coding mistake, you may receive one of the following return codes as a string.

Return code	Meaning
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The input argument is less than one or greater than the NECBS parameter
5	NUSERS = 1
8	No argument specified
9	Checkpointing is inactive, if using extended quiesce ECB named CPQZ
12	Invalid argument CPQZ or invalid argument following QZSIG.

The $ECBDGET function retrieves data set by either the $ECBDSET or $POST functions for the specified ECB.

Example

```
%X = $ECBDGET(17)
```

$ECBDSET

Function

Set string data associated with an Event Control Block (ECB)

Syntax

```
$ECBDSET({ECB-number | 'CPQZ'},{'string'})
```

Where:

- *ECB-number* is a string with a numeric value from one to the NECBS parameter that identifies the ECB in which to store the string of data. The *ECB-number* can be expressed as a numeric literal, a %variable, or a field name.
- CPQZ is a named ECB used by the NonStop/204 facility to automatically post an extended quiesce. See the *Model 204 System Manager's Guide* for an explanation of the facility.

  To use CPQZ, you need not set the NECBS parameter. CPQZ can be expressed as a literal, a %variable, or a field name. When CPQZ is specified, the value of the $ECBDSET function can be non-null (except for error values) during only extended quiesce. CPQZ is internally cleared to null when the system exits extended quiesce.
- *string* can be up to 255 bytes long. It can be a numeric, a literal, a %variable, or a field name.

  Unless explicitly reset to null, data strings persist whether or not the ECB is posted. Depending on the sequence, data strings can be changed by either the $POST or $ECBDSET functions.

Usage

The $ECBDSET function returns the following return codes:

Return code	Meaning
0	Success
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The first argument is less than one or greater than the NECBS parameter
5	NUSERS = 1
7	$ECBDSET missing data argument
8	No argument specified
9	Extended quiesce environment error; issue a $STATUSD function call for details.
11	Cannot be issued after QZSIG has been posted
12	Invalid argument CPQZ or invalid argument following QZSIG
13	Cannot be issued outside of extended quiesce

Note: The $ECBDSET function associates a string with an ECB, regardless of whether the ECB is posted or not. String data set by $ECBDSET is accessible using the $ECBDGET function.

Example

```
%X=$ECBDSET(1,’This is about managing user threads’)
```

$ECBTEST

Function

Check an Event Control Block (ECB) to see if it is posted

Syntax

```
$ECBTEST(ECB-number | ’CPQZ’ | ’QZSIG’)
```

Where:

- *ECB-number* is a string with a numeric value from one to the NECBS parameter that identifies which ECB to test for its post status. The *ECB-number* can be expressed as a numeric literal, a %variable, or a field name.
- CPQZ is a named ECB used by the NonStop/204 facility to automatically post an extended quiesce. See the *Model 204 System Manager’s Guide* for an explanation of the facility. To use CPQZ, you need not set the NECBS parameter. CPQZ can be expressed as a literal, a %variable, or a field name.
- QZSIG is a named ECB used by the NonStop/204 facility to signal when an external backup is completed. See the *Model 204 System Manager’s Guide* for an explanation of the facility. To use QZSIG, you need not set the NECBS parameter. QZSIG can be expressed as a literal, a %variable, or a field name.

Usage

Use the $ECBTEST function to obtain ECB status, posted or not, through the return code. The post code, if set by $POST, is accessible using the $STATUSD function. The following return codes apply to the $ECBTEST function:

Return code	Meaning
0	Not posted
1	Posted
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The first argument is less than one or greater than the NECBS parameter
5	NUSERS = 1

Return code	Meaning
8	No argument specified
9	Checkpointing inactive, if using extended quiesce named ECBs, CPQZ or QZSIG

Example

```
%RC=$ECBTEST(1)
```

$ECFSTAT

Function Returns the detailed completion code from the previous EXTERNAL statement.

Syntax The $ECFSTAT function returns:

```
'RC=cccc, COMPLETION CODE=tww, REASON CODE=rrrrrrrr'
```

Where Arguments can be specified as

Aurgument	Represents…
cccc	Completion code in decimal format.
t	ABEND type, either S for SYSTEM or U for USER.
ww	One of the following ABEND values: • SYSTEM values are three hexadecimal digits followed by a blank • USER values are four decimal digits
rrrrrrrr	Reason code, eight hexadecimal digits

Usage If ECF is not active or the user has not executed any EXTERNAL CALL, EXTERNAL DELETE, or EXTERNAL LOAD statements, a NULL string is returned.

$EDIT

The $EDIT function performs numeric and alphanumeric editing. This function returns a string modified according to a user-specified edit pattern. $EDIT enables you to perform the types of editing listed in Table 27-2:

Table 27-2. Editing types for $EDIT

Type	Provides the ability to edit an input string…
Numeric	In a manner similar to numeric editing in COBOL. Note the following differences: • Exponential notation, the scale position character (P), the sign character (S), and repetition notation (the use of parentheses to enclose an integer) are not supported by $EDIT. • The B character is not supported. However, a blank (a space) in the mask provides the same functionality as the B character. • A colon is supported as a simple edit character. • Overflow (to the left of the decimal point) is handled by returning a string of # characters.
Alphanumeric	By: • Inserting characters at designated points • Removing blanks from the input string • Truncating the output at designated points either conditionally or unconditionally

Syntax

The format of the $EDIT function is:

```
$EDIT(input,mask[,justification][,edit-type])
```

where:

- *input* specifies the input data to be edited. The input can be either numeric or string.
  - If numeric editing is performed, string input is converted to numerics prior to editing.
  - If alphanumeric editing is performed, numeric input is converted to a string prior to editing.
- *mask* specifies the edit mask. The mask can be either numeric or alphanumeric. The characters that can be used to define each type of edit mask are discussed in detail beginning with Table 27-3 on page 27-40.
- *justification* specifies the direction in which the input and mask are processed. Justification options are:
  - L for left-justification, the default for alphanumeric editing
  - R for right-justification, the default for numeric editing

- *edit type* specifies the type of editing. Edit type options are N for numeric editing and A for alphanumeric editing. The default edit type is N.

Alternate syntax

Alternatively, you can invoke $EDIT with this format:

```
$EDIT{A | N} (input,mask[,justification])
```

where the arguments are the same as described above, except that the edit type is specified after $EDIT, rather than after the justification argument.

Numeric edit mask

A numeric edit mask can contain up to 255 characters and have as many as 15 digit positions. Justification of the result defaults to right justification for numeric editing. Left justification causes all leading and trailing blanks to be removed from the resulting string. If the result of a numeric edit is all blanks and left justification has been requested, a null string is returned. A numeric edit mask consists of one or more of the characters listed in Table 27-3. The use of the characters within each character type is described after the table.

Table 27-3. $EDIT numeric edit mask characters

Character	Type	Description
9	Data	The position contains a number (0-9).
blank	Simple	A blank is embedded in the specified position.
0	Simple	A numeric zero appears in the specified position.
,	Simple	A comma appears in the specified position.
/	Simple	A slash appears in the specified position.
:	Simple	A colon appears in the specified position.
.	Actual decimal	A decimal point (period) appears in the position and specifies decimal alignment. A period can appear only once in a mask.
CR	Fixed	CR appears in the output string if the input argument is negative. If the input argument is positive, two spaces appear rather than CR. CR can be specified only once in a mask and must be specified in the right-most position of the mask.
DB	Fixed	DB appears in the output string if the input argument is negative. If the input argument is positive, two spaces appear rather than DB. DB can be specified only once in a mask and must be specified in the right-most position of the mask.

Table 27-3. $EDIT numeric edit mask characters (continued)

Character	Type	Description
$	Fixed or float	A dollar sign appears in the specified position in the output string for a fixed $. For a float $, the position might contain either a dollar sign, a digit, or a space depending upon the location of the float character.
+	Fixed or float	The sign of the output string with a fixed + can be either positive or negative, depending on the value of the input argument. A plus or minus sign is placed in the specified position in the output string. For a float +, a plus sign, minus sign, digit, or space might be placed in the output string depending upon the location of the float character.
-	Fixed or float	A minus sign is placed in the output string for a fixed - only Float if the value of the input argument is negative. For a float -, a minus sign, digit, or a space might be placed, depending upon the location of the float character. If the input argument is a positive value, a space replaces a minus sign.
Z	Suppress	Any leading character position that contains a zero is to be replaced by a space.
*	Suppress	Any leading character position that contains a zero is to be replaced with an asterisk.
V	Assumed decimal	The position of an assumed decimal point. The V character can appear only once in a character string.The V does not denote an actual character position; space is not reserved for it in storage.

Each character type (data, simple insertion, actual decimal, fixed, float, suppression, and assumed decimal) is discussed in detail below.

- **Data**—When a data edit character (9) is used, the numeric value in the input string replaces the appropriate data character in the mask. If the input argument has fewer characters than the mask, a 0 is returned in each unfilled position. For example:

  ```
  $EDIT(2573,’99999’)
  ```

  returns the value:

  ```
  02573.
  ```

- **Simple insertion**—When a simple insertion character (a blank, zero, comma, slash, or colon) is used, the specified mask positions are reserved and replaced by the appropriate character in the output string. For example:

  ```
  $EDIT(2573478977,’9,999,999,999’)
  ```

  returns the value:

```
2,573,478,977.
```

- **Actual decimal**—When the actual decimal character (the period) is used, the specified decimal position is reserved and placed in the output string and the input string is decimal aligned. For example:

  ```
  $EDIT(25.734,'99999.99')
  ```

  returns the value:

  ```
  00025.73.
  ```

  Note: If the mask contains fewer digits to the right of the decimal point than the input argument, the input argument is scaled by dropping the excess digits to the right of the decimal point.

- **Fixed**—When a fixed edit character (CR, DB, $, +, -) is used, the specified leading or trailing position is reserved and replaced with the appropriate character in the output string. A fixed edit character can occur only once in a mask. For example:

  ```
  $EDIT(-457.22,'$999.99CR')
  ```

  returns the value:

  ```
  $457.22CR.
  ```

  Consider the following when using fixed edit characters:

  – A mask can contain at most one fixed sign character (+, -, CR, or DB).

  – If the edit mask contains a trailing fixed sign character and that character is set to blanks in the result, right justification causes the trailing blank to be retained.

  – When a + or - is specified, it must be the first or last character in the mask.

  – No more than one $ and one +, -, CR, or DB can be specified within a mask. The $ must either be specified in the left-most position of the mask or be preceded only by a + or -.

- **Float**—When float edit characters ($, +, or -) are used, the specified positions are reserved in the output string. (A float mask must contain at least two consecutive float characters of the same type.) Float characters can be specified in any leading character position to the left of the decimal point. Float characters also can be specified in any trailing character position to the right of the decimal point only if all digit positions are represented by the float character.

  The float character is inserted immediately to the left of either the first significant digit, a digit position specified with a 9, or a decimal point, whichever is furthest to the left. All positions to the left of the inserted float character are replaced with blanks. For example:

  ```
  $EDIT(-8283.56,'$$$$$$999.99CR')
  ```

  returns the value:

```
$8283.56CR.
```

Note the following considerations when using float edit characters:

- Floating $, +, and - characters are mutually exclusive within the mask.
- The float mask can be interrupted by one or more simple characters or by an assumed or actual decimal point. Simple insertion characters to the left of the floating character actually used in the output string are converted to blanks in the output string. For example:

  ```
  $EDIT(-123456.789, '++,+++,999.99')
  ```

  returns a value of:

  ```
  -123,456.78.
  ```

- If float characters appear in all the numeric data positions and the value of the input argument is zero, the output string contains all spaces. For example:

  ```
  $EDIT(0000.00, '$$,$$$,$$$.$$')
  ```

  returns a null string.

• **Suppression**—When a suppression character (Z or *) is used, the specified position is reserved and replaced with the appropriate character in the output string. One or more suppression characters can be specified in a mask; however, Z and * are mutually exclusive.

A suppression character can be specified in any leading character position to the left of the decimal point. A suppression character also can be specified in any trailing character position to the right of the decimal point only if all positions to the right are represented by the suppression character.

$EDIT returns the value of the suppression character (a blank for each Z or an asterisk for each *) for each leading position that has a value of 0, stopping at the left-most position that either contains nonzero data or is an actual decimal point.

For example:

```
$EDIT(+84783.56,'$Z,ZZZ,ZZZ.ZZ')
```

returns the value $84,783.56. Note that any simple edit characters (a blank, zero, comma, slash, or colon) to the left of the stopping point are replaced with the suppression character (blank or *).

In addition, if all data positions in the mask are represented by suppression characters and the value of the input is zero, all characters in the mask (including simple insertion characters) are replaced with the suppression character. For example:

```
$EDIT(0,'$ZZ,ZZZ')
```

returns all blanks.

- **Assumed decimal**—When the assumed decimal character is used, the input is decimal aligned. However, the decimal does not occupy an output position. For example:

  ```
  $EDIT(.12345,’999V 99’)
  ```

  returns the value:

  ```
  000 12.
  ```

  Note these additional considerations when defining a numeric edit mask:

  – The mask must contain at least one occurrence of 9, Z, or * or more than one occurrence of +, -, or $.

  – All characters except the assumed decimal character (V) are counted in the size of the output string.

  – The special character (.) and the assumed character (V) are mutually exclusive within a mask.

  – The $ fixed character and the + or - floating characters are mutually exclusive.

  – Floating edit characters and suppression characters are all mutually exclusive.

  – Floating edit characters and suppression characters to the left of an actual or assumed decimal point cannot be preceded by the data edit character (9). A suppression character to the left of an actual or assumed decimal point also cannot be preceded by a float character.

  – The mask should be large enough to accommodate the input string or truncation occurs. If the mask contains fewer digits to the right of the decimal point than the input argument, the input argument is scaled by dropping the excess digits to the right of the decimal point.

Alphanumeric edit mask

An alphanumeric edit mask can contain up to 255 characters and as many as 255 edit character positions. Justification of the result defaults to left justification for alphanumeric editing. If the input string is a null string, the mask is still processed. If the string is longer than the number of character positions indicated in the mask, the excess characters in the input string are ignored. An alphanumeric edit mask can consist of one or more of these character types:

Character	Used in alphanumeric edit masks…
Insertion	Any character can be an insertion character. However, if the insertion character is not a letter (A–Z, a–z), a digit (0–9), or a blank, it must be preceded by an escape character.

Character	Used in alphanumeric edit masks…
Escape (!)	The escape character indicates that the next character in the edit mask is interpreted literally (as an inserted character rather than as a selection character). The escape character must precede non-alphanumeric insertion characters. • If the justification argument is R (processing from right to left), the escape character must be to the right of the insertion character. • If the justification argument is L (processing from left to right), the escape character must be to the left of the insertion character. • If the escape character is the last character in the mask, it is ignored. • If the escape character is omitted before a non-alphanumeric insertion character, the mask is considered invalid, a null result is returned, and an error message is issued.
Simple selection	The characters listed in Table 27-4 are considered simple selection characters:
Special selection	The characters listed in Table 27-5 are considered special selection characters.

Table 27-4. Simple selection characters

Character	The placement of…
+	Single input character. If no input characters remain, a blank is inserted and processing continues with the next character in the mask.
*	All remaining input characters. If no input characters remain, a blank is not inserted and processing continues with the next character in the mask.
¬	Next non-blank character in the input string. If no input characters remain, a blank is not inserted and processing continues with the next character in the mask.

Table 27-5. Special selection characters

Character	Description
<	Whether to continue processing or truncate the result, depending on processing direction. (See the discussion that follows for details concerning processing direction.)
)	The placement of a single character. If no input characters remain, a blank is inserted and processing of the mask is terminated.
-	The placement of a single character. If no input characters remain, a blank is inserted and all subsequent insertion characters are replaced with blanks.

When a special selection character is used, truncation is handled in the following manner:

- The processing direction (in the direction specified by the justification option) determines how the < or > character is handled:

  When a < or > character points in the processing direction, it acts as a continuation character and is handled as follows:

  – The next remaining input character is placed in the result and the next character in the mask is processed.

  – For no remaining input characters, the result is unaffected and the next character in the mask is processed.

  If a < or > points in the direction opposite from the processing direction, it acts as a truncation character and is handled as follows:

  – For several remaining input characters, the next input character is placed in the result and the next character in the mask is processed.

  – For one remaining input character, that character is placed in the result and the result is truncated after the character.

  – For no remaining input characters, the result is truncated after the previous result character.

- The | character has the following effect:

  – For several remaining input characters, the next input character is placed in the result and the next character in the mask is processed.

  – For one remaining input character, that character is placed in the result and the result is truncated after the character.

  – For no remaining input characters, a blank is placed in the result and the result is truncated after the blank.

- The _ (underscore) character has the following effect:

  – If there are several remaining input characters, the next input character is placed in the result and the next character in the mask is processed.

  – If there is only one remaining input character, that character is placed in the result and all subsequent insertion characters are replaced with blanks.

  – If there are no remaining input characters, a blank is placed in the result and all subsequent insertion characters are replaced with blanks.

Example 1

This example illustrates the use of a right-justified edit mask to format a phone number with an area code:

```
$EDIT(6171234567,’(!+++)! >++-!++++’,’R’,’A’)
```

results in the following output:

```
(617) 123-4567
```

Example 2

If the same edit mask were used for a string that did not contain an area code, as shown below:

```
$EDIT(2344567,'(!+++)! >++-!++++','R','A')
```

the following output would result:

```
234-4567
```

Example 3

This example illustrates the use of a left-justified edit mask for formatting phone numbers, some of which have extensions of two, three, or four digits:

```
$EDIT(%PHONE,'!(+++!) +++!-+++  EXT ++++',,'A')
```

The value of the input argument and the output that would result from using the preceding example are listed in this table:

Input	Output
`61712345671111`	`(617) 123-4567 EXT 1111`
`2122344567`	`(212) 234-4567`
`516765432134`	`(516) 765-4321 EXT 34`

Error conditions

Certain errors that can occur when using $EDIT produce the following results:

- A null string is returned if the input argument or edit mask is omitted, or if the edit mask is invalid, or if data is incompatible with the edit type (for example, alpha data when the edit mask is numeric).
- The default for the type of editing indicated by the edit mask is used if the justification option is invalid.
- A string of # characters (with a length equal to the number of output positions in the edit mask) is returned if a numeric editing overflow occurs to the left of the decimal point.

Other error conditions for $EDIT generate a Model 204 error message.

$EFORMAT

The $EFORMAT function converts numeric values to exponent notation. For a detailed discussion of exponent format refer to “Exponent notation” on page 4-9.

Syntax

The format of the $EFORMAT function is:

```
$EFORMAT(value,significant digits,fractional digits)
```

where arguments are:

- *value* argument must contain the numeric value to be formatted. If this argument is passed to $EFORMAT with a nonnumeric or invalid value, $EFORMAT returns a zero.
- *significant digits* argument must contain a positive integer identifying the number of significant digits to print. The default is 15. If this argument is passed to $EFORMAT with an invalid value, $EFORMAT returns a null string.
- *fractional digits* argument must contain a positive integer identifying the number of significant digits to print to the right of the decimal point. The number of digits specified is always printed to the right of the decimal point, even if the digits are zero.

  If this argument is omitted, all significant digits are printed with one digit placed to the left of the decimal point in the form *hn.nn* through *nEnn*. If this argument is passed to $EFORMAT with an invalid value, $EFORMAT returns a null string.

Example

```
BEGIN
GET.FLOAT:  FIND ALL RECORDS FOR WHICH
               FLOATFLD1 = 3444300E15
            END FIND
            FOR EACH RECORD IN GET.FLOAT
               PRINT 'E FORMAT NUMBER IS ' WITH -
                  $EFORMAT(FLOATFLD1,5,2)
            END FOR
END
```

results in this output:

```
E FORMAT NUMBER IS 344.43E19
```

$ENCRYPT

The $ENCRYPT function performs a one-way encryption of a zero to eight-character string. Longer strings are truncated at eight characters. The result of this function is an eight-byte encrypted representation of the input string.

Syntax

The format of $ENCRYPT is:

```
$ENCRYPT(string,parameter)
```

$ENCRYPT accepts the second argument (*parameter*) so that the same source string can be encrypted to different strings in different systems.

Note: You should use a parameter value of less than 1000 for best performance; numbers significantly larger than 1000 can cause serious performance degradation.

Example

```
$ENCRYPT ($READINV ('ENTER PASSWORD'),%PARAM)
```

returns one encrypted value if %PARAM equals 28 and another value if %PARAM equals 18. If no parameter is supplied, a default of 15 is used. A null string is returned and an error message is issued if a parameter of zero or a negative value is specified.

$ENTER

The $ENTER function provides efficient terminal dialogue with users of data entry applications.

Syntax

The format of the $ENTER function is:

```
$ENTER (number of values, prompt string,
        default value, separator, name prefix)
```

$ENTER prompts the terminal operator with the value entered in the prompt string argument. If this argument is omitted, this prompt appears:

```
ENTER DATA
```

The operator enters a single line of input which is read by Model 204. This line is parsed into individual values using the value of the separator argument as the delimiter.

- The *separator* argument must be a single character. If it is omitted, a comma is used as the default separator. If an input value is longer than 255 characters, a warning message is issued and the value is truncated. The input values are assigned in the order of %variables using the value of the name prefix argument as follows:

  ```
  %name prefix 01, %name prefix 02, %name prefix
  03,…
  ```

- If the *name prefix* argument is omitted, the %variables are named %01, %02, %03, and so on. All %variable names generated by $ENTER must appear elsewhere in the request so that the variables can be allocated during compilation.

- The *number of values* argument is required; it indicates the number of values assigned by $ENTER. If any values are omitted, the value specified in the default value argument is assigned to the appropriate %variables. If the default value argument is omitted, a single blank character is the default.

If $ENTER completes successfully, it returns a value of 0. If it fails for any of these reasons, $ENTER returns a value of 1 and prints an explanatory message:

- Too many values are included on the input line.
- The number of values argument is less than 1 or greater than 99.
- Required %variables were not allocated during compilation (a %variable used in $ENTER does not appear elsewhere in the request).

Example 1

Suppose a request contains the following function call:

```
%X = $ENTER(3, ’ENTER 3 VALUES’, ’99999’ ,, ’ZZ’)
```

The following prompt is displayed at the terminal:

```
ENTER 3 VALUES
```

The user enters:

```
A1,,C3
```

$ENTER returns a value of 0. This result is equivalent to the following:

```
%ZZ01 = ’A1’
%ZZ02 = ’99999’
%ZZ03 = ’C3’
```

Example 2

```
%Y = $ENTER (4,,,’/’,)
```

This default prompt is displayed:

```
ENTER DATA
```

The user enters:

```
ONE/TWO/THREE/FOUR
```

$ENTER returns a value of 0, which is equivalent to:

```
%01 = ’ONE’
%02 = ’TWO’
%03 = ’THREE’
%04 = ’FOUR’
```

$ERRCLR

The $ERRCLR function clears the error message text returned by the $ERRMSG and $FSTERR functions. The $ERRCLR function takes no argument.

Example

```
%DUMMY=$ERRCLR
```

$ERRMSG

The $ERRMSG function returns a variable length string of up to 79 characters containing the prefix and text of the last counting error message or request cancellation message received by the user.

A null value is returned if no counting error or request cancellation message has been received since the beginning of the user's Model 204 session, or since the last call to the $ERRCLR function. Refer to the *Model 204 Messages Manual* for more information on counting messages. This function takes no arguments.

Example

A sample $ERRMSG function with an ON ERROR unit follows. To learn about ON ERROR units, see "ON units" on page 12-21.

```
BEGIN
ERROR.PROC: ON ERROR
               PRINT 'THE REQUEST IS ENDING'
              PRINT 'THE LAST ERROR MESSAGE RECEIVED
WAS:'
               PRINT $ERRMSG
            END ON
GET.RECS:   FIND ALL RECORDS FOR WHICH
               AGENT = CASOLA
            END FIND
            FOR EACH RECORD IN GET.RECS
               .
               .
               .
            END FOR
END
```

$FDEF

The $FDEF function lets you access the attributes of a field from within a Model 204 procedure. $FDEF maps the attributes of a field, whose values can then be read via an image similar to the ZFIELD image described on page 27-52. Unlike $DSCR, which $FDEF supplants, the attributes are displayed in a readable fashion without parsing. $FDEF works only for files (not groups).

Syntax

The format for the $FDEF function is:

```
%image:item = $FDEF([FILE] filename [AT location],
fieldname)
```

where:

- *%image:item* is the image item to which $FDEF returns the data.
- *filename* is a %variable or a literal name of the file. A file synonym name can also be used. When *filename=groupname* the $FDEF function assumes that the name passed is a file name, not a group name.
- *location* is the name of the remote node where the file is located.
- *fieldname* is a %variable or a literal name of the field to be described.

If the field specified in the $FDEF argument is not defined in the opened file specified by filename, $FDEF returns a U (undefined) in the second image item (DEFER.Y_N). If the file is not open, a U is returned in the second (DEFER.Y_N) image item and an N is returned in the third (FRV.Y_N) image item.

The field code, which is returned in the 21st image item, is unique for each field within a file. However, this value cannot remain constant for any one field over time, and it cannot be the same for the same field name in different files. Field names do not always hash to the same field codes because of deleted fields and hash collisions. The field code which is returned in the ZFIELD image is the same as that which appears in CCAJRNL RECTYPE=6 entries. This information is valuable for CCA support in case you need to run REGENERATE but cannot because you are missing one or more CCAJRNL datasets since your last DUMP of the file.

The ZFIELD image

An image is required by the $FDEF and $LSTFLD functions. The ZFIELD image, which can give you complete field attribute information (see the $FDEF example below), is provided on the Model 204 installation tape by CCA.

Note: When using ZFIELD, be aware that $FDEF output maps to the image of ZFIELD. Therefore, *do not change the order of the image items* in ZFIELD.

The location of ZFIELD for your site is listed in this table:

IF your site runs under...	THEN the ZFIELD image is stored in...
z/OS	The JCL library

IF your site runs under...	THEN the ZFIELD image is stored in...
VM	The 2nd tape file (193) as an EXEC
VSE	The JCL library

Example

In the example on the next page, you provide a field name and Model 204 displays output that indicates if the field is KEY or preallocated. If the field is preallocated, Model 204 also displays the number of occurrences and the length of the field. This procedure maps the $FDEF output to the ZFIELD image (shown on the following page). Therefore, if the field is KEY, then the KEY.Y_N image item contains a ‘Y’.

```
PROCEDURE DISPFLD
OPEN DAILY
OPENC VEHICLES
XXXX
BEGIN
%FIELD IS STRING LEN 50
%FIELD = $READ('ENTER FIELDNAME')
*                                                    *
* include the ZFIELD proc and prepare the ZFIELD
image *
*                                                    *
INCLUDE ZFIELD
PREPARE IMAGE ZFIELD
*                                                    *
* use the FDEF image item in ZFIELD to check the
fieldname*
*                                                    *
%ZFIELD:FDEF = $FDEF('VEHICLES',%FIELD)
IF %ZFIELD:KEY.Y_N = 'Y' THEN
   PRINT %FIELD ' IS KEY'
ELSE
   PRINT %FIELD ' IS NOT A KEY FIELD'
END IF
IF %ZFIELD:OCCURS > '0' THEN
   PRINT %FIELD ' IS PREALLOCATED, WITH ' -
      %ZFIELD:OCCURS ' VALUES'
   PRINT 'THE LENGTH OF ' %FIELD ' IS '
%ZFIELD:LENGTH
ELSE
   PRINT %FIELD ' IS NOT PREALLOCATED'
END IF
END
```

The following is the output produced by the procedure DISPFLD for the field DEDUCTIBLE:

```
INCLUDE DISPFLD
```

```
??ENTER FIELD NAME
DEDUCTIBLE
DEDUCTIBLE IS KEY
DEDUCTIBLE IS PREALLOCATED WITH 1 VALUES
THE LENGTH OF DEDUCTIBLE IS 3
```

ZFIELD image for $FDEF and $LSTFLD

The following image can be used by the $FDEF and $LSTFLD functions. If you write your own image, be aware that $FDEF maps to the locations of the image items, rather than the names.

```
IMAGE ZFIELD
NAME                 IS STRING LEN 255
DEFER.Y_N            IS STRING LEN 1
FRV.Y_N              IS STRING LEN 1
KEY.Y_N              IS STRING LEN 1
MANY.VALUED.Y_N      IS STRING LEN 1
CODED.Y_N            IS STRING LEN 1
STRING.Y_N           IS STRING LEN 1
NUMERIC.RANGE.Y_N    IS STRING LEN 1
INVISIBLE.Y_N        IS STRING LEN 1
SECURED.Y_N          IS STRING LEN 1
UPDATE.IN.PLACE.Y_N  IS STRING LEN 1
OCCURS.Y_N           IS STRING LEN 1
FLOAT.Y_N            IS STRING LEN 1
ORD.NUM.Y_N          IS STRING LEN 1
ORD.CHAR.Y_N         IS STRING LEN 1
PURE.DBCS.Y_N        IS STRING LEN 1
MIXED.DBCS.Y_N       IS STRING LEN 1
UNIQUE.Y_N           IS STRING LEN 1
OCCURS.ONCE.Y_N      IS STRING LEN 1
FUTURE.EXPANSION     IS STRING LEN 36
FIELD.CODE           IS BINARY LEN 4 UNSIGNED
ORDERED.Y_N          IS STRING LEN 1
LENGTH               IS BINARY LEN 1 UNSIGNED
LEVEL                IS BINARY LEN 1 UNSIGNED
LRESERVE             IS BINARY LEN 1 UNSIGNED
NRESERVE             IS BINARY LEN 1 UNSIGNED
SPLITPCT             IS BINARY LEN 1 UNSIGNED
NO.OF.IMMEDIATES     IS BINARY LEN 1 UNSIGNED
OCCURS               IS BINARY LEN 1 UNSIGNED
PAD.CHAR             IS STRING LEN 1
FDEF                 IS STRING LEN 67 AT DEFER.Y_N
BIN1                 IS BINARY LEN 4 UNSIGNED
BIN2                 IS BINARY LEN 4 UNSIGNED
BIN3                 IS BINARY LEN 4 UNSIGNED
BIN4                 IS BINARY LEN 4 UNSIGNED
LOOPVAR              IS STRING LEN 16 AT BIN1
END IMAGE
```

$FLDLEN

The $FLDLEN function interprets its character string argument as a field name.

- In file context, $FLDLEN returns a number representing the LENGTH of the field if the field has a LENGTH attribute in its description. In file context, $FLDLEN returns 0 if the specified field does not have the LENGTH attribute in its description.
- In group context, $FLDLEN returns the minimum LENGTH specification included for all of the files. In group context, $FLDLEN returns 0 only if the field is described without a LENGTH attribute in every file in the group.
- If the field specified as the $FLDLEN argument is not defined in the current file or group, $FLDLEN returns a value of -1.

Example

```
$FLDLEN ('SOCSECNO')
```

equals 9 if the description of SOCSECNO for the file contains LENGTH 9.

$FLOAT

The $FLOAT function assigns a 4-byte floating point number to a 4-byte string, without any conversion. This function is intended for use in writing a floating point value to a USE dataset. The $FLOAT function takes one argument containing a numeric value. If the argument is omitted, a value of 0 is returned.

Model 204 maintains 15 significant decimal digits of precision for 8-byte floating-point numbers and 6 significant digits of precision for 4-byte floating-point numbers. For an expanded discussion of rounding numbers, please refer to the section “Mapping and precision adjustment” on page 31-3.

Note: CCA does not recommend using $FLOAT in new applications, although this function is supported for compatibility reasons.

$FLOATD

The $FLOATD function assigns an 8-byte floating point number to an 8-byte string, without any conversion. Like $FLOAT, this function is intended to be output to a USE dataset. The $FLOATD function takes one argument containing a numeric value.

Note: CCA does not recommend using $FLOATD in new applications, although this function is supported for compatibility reasons.

$FLSACC

The $FLSACC function, combined with the $FLSCHK function, allows a User Language request to check for field-level security access violations before they occur. This reduces evaluation-time errors and request cancellations, and it helps to ensure that the files being updated are not left in a logically inconsistent state (“Resolution of field types and levels” on page 10-33”).

$FLSACC can determine a user’s access rights to a particular field or to all fields in a file or group. The function returns a character string representing the user’s access rights to the specified field(s). If the field is not defined in the file or group, a null string is returned. The string normally contains a combination of the characters listed in this table:

Character	Meaning
S	SELECT access rights
R	READ access rights
U	UPDATE access rights
A	ADD access rights

Syntax

The format of the $FLSACC function is:

```
$FLSACC (fieldname [,name])
```

where:

- *fieldname* is a character string that is interpreted as the name of the field whose access is being checked.

  If this argument is omitted or null, every field in the file or group specified in the *name* argument is checked. In this case, the output string returned by $FLSACC contains an access indication (S, R, U, A) only if that level of access applies to every field in the file or group. This use of $FLSACC is costly system overhead because all field descriptions must be examined.

- *name* is optional; use it to control the file or group context of the function. The format for this argument is:

  ```
  [[FILE | [PERM | TEMP] GROUP] name [AT location]
    | $CURFILE | $UPDATE]
  ```

  If access for the current file of a FOR loop is desired, $CURFILE can be used as the second argument. $UPDATE can be used to indicate the name of the update file in the current group. $CURFILE and $UPDATE are described in detail in “$CURFILE and $UPDATE functions” on page 16-29. For single file context, the field level security access rights are well defined. If a field name is specified in group context, $FLSACC returns the maximum access rights for the group. If the *name* argument is omitted or null, the context used for the check is the context of the statement that contains the function.

$FLSCHK

The $FLSCHK function, combined with the $FLSACC function, allows a request to check for field-level security access violations before they occur. $FLSCHK is designed for use with the IF statement and can determine whether a given set of field-level security accesses is valid for a specified field or for all fields in a file or group. The function returns a 1 if the specified set of accesses is valid.

Syntax

The format of the $FLSCHK function is:

`$FLSCHK (fieldname, access [,filename])`

- *fieldname* is a character string representing the name of the field whose access is to be checked.

  If the field name is omitted or null, every field in the file or group is checked; the function returns a 1 only if the access is allowed for every field in the file or group. This use of $FLSHCK can be costly system overhead since all field descriptions must be examined.

- *access* is the desired access or set of accesses; the argument can include:
  - S (SELECT)
  - R (READ)
  - U (UPDATE)
  - A (ADD)

  If the access argument contains an invalid character (not S, R, U, or A), a warning message is issued and 0 is returned. If a set of accesses is specified, such as SR, a 1 is returned only if all accesses in the set are allowed for the indicated field or fields.

- *filename* is optional and can be used to control the file or group context of the function. The format for the *filename* argument is identical to the *name* argument described earlier for the $FLSACC function. For single file context, the field level security access rights are well defined. In group context, this function returns a 1 if the indicated access would compile without error. If access for the current file of a FOR loop is desired, $CURFILE can be used as the third argument. If this argument is omitted or null, the context used for the check is the context of the statement containing the function.

$FSTERR

The $FSTERR function returns a variable length string of up to 79 characters that contains the prefix and the first counting error message or request cancellation message you received since the last time that the count was reset to zero. If Model 204 has not received a counting error or request cancellation message since the beginning of your Model 204 session, or since the last call to the $ERRCLR function, $FSTERR returns a null value.

For more information about counting errors or request cancellation messages, refer to the *Model 204 Messages Manual*. To return the most recent error message, refer to the $ERRMSG function. $FSTERR is not available for Host Language Interface applications. The $FSTERR function takes no arguments.

The $FSTERR function requires an additional 88 bytes in the fixed portion of the server.

Example

An example of the $FSTERR function with an ON ERROR unit follows. For more information, refer to “ON units” on page 12-21.

```
BEGIN
FSTERR.PROC:   ON ERROR
                  PRINT ’THE REQUEST IS ENDING’
                 PRINT ’THE FIRST ERROR MESSAGE WAS:’
                  PRINT $FSTERR
               END ON

GET.RECS:      FIND ALL RECORDS FOR WHICH
                  AGENT = BLAKE
               END FIND
               FOR EACH RECORD IN GET.RECS
               .
                .
                END FOR
END
```

$GETG

The $GETG function retrieves information stored by a $SETG function in the same request or an earlier request.

The $GETG function takes a global variable name as its one argument. The global variable table is searched for a variable with the name given by the argument. The value of the function is the value of the global variable if it is found or a null string (two quotes with no space between them) if it is not found. $GETG (‘X’) does not distinguish between the following situations:

- There is no variable with the name X in the table.

- There is a variable in the table with the name X whose value is a null string (a $SETG (‘X’,’’) that had been executed earlier).

Refer to “Using global string variables to tailor a request” on page 20-13 for a detailed explanation of global variables and examples of the $GETG function within a request.

$GETL

The $GETL function returns the line number of the current line on the page on the user’s terminal or on the output dataset specified by a USE command. The first line on a page is line number 1 and is the line on which header 0 appears. The current line is the line actually being printed if $GETL is invoked:

- From a PRINT statement
- After a PRINT statement that ends with an ellipsis (…).

Otherwise, the current line is the next line to be printed. The current line becomes line 1 as soon as the bottom line of a page has been printed. $GETL takes no arguments.

$GETP

The $GETP function returns the page number currently on the user’s terminal or on the output dataset specified by a USE command. When the current output device is the normal device (no USE command is in effect), the value returned by $GETP matches the current value of the OUTPNO parameter, which indicates the current output page number. Otherwise, the value returned by $GETP is the current page number on the USE dataset. $GETP takes no arguments.

$GRMLOC

The $GRMLOC function returns the location of a missing group member.

See the *Parallel Query Option/204 User’s Guide* for more information on the $GRMLOC, $GRMNAME, $GRNLEFT, and $GRNMISS functions.

$GRMNAME

The $GRMNAME function returns the file name of a missing group member.

$GRNLEFT

The $GRNLEFT function returns the number of optional files that might fail before the value of the MAXFAIL parameter is exceeded.

$GRNMISS

The $GRNMISS function returns the number of missing group members.

$GROUPFILES

The $GROUPFILES function returns the number of files in an open group.

Syntax

```
$GROUPFILES(file-or-group-context)
```

Where:

file-or-group-context is a character string argument representing a group name.

Example

```
BEGIN
%GROUP-CONTEXT_STRING=’PERM GROUP MYGROUP’
%X=$GROUPFILES(%GROUP_CONTEXT_STRING)
PRINT ‘THERE ARE' WITH %X WITH ‘ FILES IN THE GROUP'
END
```

Produces the following output:

```
THERE ARE 5 FILES IN THE GROUP
```

Usage

- Called with a character string argument representing a group name. Supports PERM/TEMP/GROUP/FILE/AT keywords. Supports file synonyms.
- Returns the number of files in the group, or 0 if error or file context.

$HPAGE

The $HPAGE function returns a string of special characters whose length is equal to the value specified as the $HPAGE argument. $HPAGE is usually used in a line replacing header 0 (see “Formatting page headers and trailers” on page 6-14). When you use $HPAGE in a SET HEADER or SET TRAILER statement, it is replaced by the current page number when the header or trailer is printed.

Example

This $HPAGE statement:

```
SET HEADER 1 ’AUDIT REPORT’ WITH ’PAGE’ TO -
           COLUMN 30 WITH $HPAGE (2) TO COLUMN 33
```

produces a header in the format:

```
AUDIT REPORT             PAGE 13
```

The number 13 is generated by Model 204. To set the value of the page number to 0 or another value, use the $SETP function.

$HSH

The $HSH function lets you convert a string into a hash value, which is a distinct numeric representation of a given string value. You can use $HSH to build a memory-resident hash table to be used to join on keys without sorting and merging.

Syntax

The format for the $HSH function is:

```
%variable = $HSH('string')
```

where:

- *%variable* is a %variable within a User Language procedure
- *string* is an alphanumeric string or a string referenced by a VALUE IN statement.

Example

This example is continued on the next page:

```
BEGIN
*
* INPUT CUSTOMER CONTROL DATA IMAGE:
*
IMAGE CUST_REC
   NAME IS STRING LEN 20
   RATE IS STRING LEN 8
   DISCOUNT IS STRING LEN 8
END IMAGE
*
* MEMORY-RESIDENT HASH TABLE ARRAY:
*
IMAGE CUST_ARRAY
   ARRAY OCCURS 500
     NAME IS STRING LEN 20
     RATE IS FLOAT
     DISCOUNT IS FLOAT
   END ARRAY
END IMAGE
*
PREPARE IMAGE CUST_REC
PREPARE IMAGE CUST_ARRAY
```

```
*
* OPEN INPUT DATASET
*
OPEN DATASET CUSTINFO FOR INPUT
*
READ IMAGE CUST_REC FROM CUSTINFO
REPEAT WHILE $STATUS = 0
*
*  INDEX INTO HASH TABLE IS REMAINDER OF HASH OF KEY
/ SIZE OF
*  TABLE
*
   %IDX = $MOD($HSH(%CUST_REC:NAME),500)
   IF %CUST_ARRAY:NAME(%IDX) EQ '' THEN
      %CUST_ARRAY:NAME(%IDX) = %CUST_REC:NAME
      %CUST_ARRAY:RATE(%IDX) = %CUST_REC:RATE
     %CUST_ARRAY:DISCOUNT(%IDX) = %CUST_REC:DISCOUNT
   ELSE
*
*     HAVE A HASH CONFLICT, RETRY A FINITE NUMBER OF
TIMES
*
      FOR %RETRY FROM 1 TO 5
         %IDX = $MOD($HSH(%IDX),500)
IF %CUST_ARRAY:NAME(%IDX) EQ '' THEN
            %CUST_ARRAY:NAME(%IDX) = %CUST_REC:NAME
            %CUST_ARRAY:RATE(%IDX) = %CUST_REC:RATE
            %CUST_ARRAY:DISCOUNT(%IDX) =
%CUST_REC:DISCOUNT
            JUMP TO READNEXT
         END IF
      END FOR
      PRINT 'MAKE THE HASH TABLE LARGER TO AVOID
CONFLICTS'
      STOP
   END IF
READNEXT:
   READ IMAGE CUST_REC FROM CUSTINFO
END REPEAT
*
*  NOW FIND ORDER RECORDS, NO NEED TO SORT
*
FDORD: IN ORDERS FIND ALL RECORDS WHERE -
                 ORD DATE IS NUM IN RANGE FROM 89120
TO 89150
                   STATUS = 'SHIPPED'
        END FIND

FORORD: FOR EACH RECORD IN FDORD
           %IDX = $MOD($HSH(CUST NAME))
           IF %CUST_ARRAY:NAME(%IDX) EQ CUST NAME
THEN
              CALL PRINT_INVOICE
           ELSE
```

```
*
*        HAVE A HASH CONFLICT, RETRY A FINITE NUMBER
OF TIMES
*
           FOR %RETRY FROM 1 TO 5
              %IDX = $MOD($HSH(%IDX),500)
             IF %CUST_ARRAY:NAME(%IDX) EQ CUST NAME
THEN
                 CALL PRINT_INVOICE
                 JUMP TO NEXTREC
              END IF
           END FOR
          PRINT 'CUSTOMER NAME NOT FOUND =' AND CUST
NAME
NEXTREC:
        END FOR
END
```

$INCRG

The $INCRG function performs simple arithmetic on global variables. Global variables are discussed in Chapter 20.

Syntax

The format of the $INCRG function is:

$INCRG(*global name* [,*increment*][,*decimal_places**)

where:

- *global name* must be specified as a string.
- *increment* is an optional numeric value; the default value is 1.
- *decimal places* is optional and specifies the number of decimal places to be preserved in the result of the operation. The fractional part of the result is truncated (not rounded) or padded with zeros as appropriate. If this argument is omitted, the $INCRG result has the same number of decimal places as the original global value.

How $INCRG works

$INCRG adds the value specified in the increment argument to the global variable specified in the global name argument and returns a completion code. The $INCRG operation preserves up to 15 digits of precision. Both the global value and the increment are treated as signed quantities. Thus, if the increment is less than 0, the function performs subtraction.

Model 204 converts the global value set by $INCRG to floating point and then converts it back to a string. Thus, any leading and trailing zeros, plus signs, and internal spaces are lost. The decimal point is also lost unless it is followed by a nonzero decimal digit before the 15th digit place. If the global variable does not exist, $INCRG creates one with a value equal to the increment. If a value is not specified as the *decimal_places* argument in the function call, the created global variable have no decimal places.

$INCRG returns a completion code indicating the success or failure of the operation, along with an explanatory message. Possible codes are:

Code	Meaning
0	$INCRG completed normally
1	Global value is not numeric
2	No room for new value of global (size of global can change after increment)

IF statements

In an IF statement of the form “IF expression”, the THEN clause is evaluated if expression has any value other than 0. Thus, IF $INCRG (‘X’) THEN JUMP TO ERROR.ROUTINE transfers control to the ERROR.ROUTINE statement on either of the two possible error conditions. If different actions are to be taken for each distinct condition, this technique is recommended:

```
JUMP TO (ERROR.TYPE1,ERROR.TYPE2) $INCRG (’X’)
```

In this case, no jump is taken if $INCRG completes successfully and returns 0.

Example 1

```
PRINT $INCRG (’GLOBAL’)
```

prints 0 if the operation is successful and increases the value of GLOBAL by 1.

Example 2

```
PRINT $INCRG (’X’, -.5)
```

prints 0 if the operation is successful and decreases the value of global X by.5.

$INDEX

$INDEX compares two character strings and returns a number equal to the first position within the first string at which the second string appears.

Example 1

```
$INDEX (’OTHER’, ’THE’) equals 2
$INDEX (’SAME’, ’SAME’) equals 1
```

Example 2

If the second string is not contained in the first or is a null string, zero is returned:

```
$INDEX ('SAME', 'OTHER') equals 0
$INDEX ('SOME', 'SOMEMORE') equals 0
$INDEX ('ABC', '') equals 0
```

Example 3

This request uses $INDEX to separate the last name from a field containing a full name in the form "last name, first name":

```
BEGIN
FIND.RECS: FIND ALL RECORDS FOR WHICH
               DATE OF BIRTH IS LESS THAN 660000
           END FIND
           FOR EACH RECORD IN FIND.RECS
               %LASTNAME = -
                   $SUBSTR (FULLNAME, 1, -
                            $INDEX (FULLNAME,',')-1)
               ADD LAST NAME = %LASTNAME
                   .
                   .
                   .
```

$ITSOPEN

The $ITSOPEN function lets you determine whether a file is open. $ITSOPEN only checks files, not groups of files. The return codes are:

0	the file is *not* open
1	the file is open

Syntax

The format for the $ITSOPEN function is:

$ITSOPEN({[FILE] *name* [AT *location*]

| [PERM | TEMP] [GROUP] *name*})

where *name* (optional) is a %variable or a literal name of the file or group. You *must* enter the filename in uppercase. A file synonym name can also be used. If you do not enter a location (specifying a null argument), Model 204 uses the reference context (at compile time) of the statement which calls the function.

Example

```
BEGIN
```

```
%FILE IS STRING
%FILE = $READ('ENTER THE FILENAME')
IF $ITSOPEN(%FILE) THEN
     PRINT %FILE ' IS OPEN'
ELSE
     PRINT %FILE ' IS NOT OPEN'
END
```

$ITSREMOTE

The $ITSREMOTE function (valid in PQO only) lets you determine whether a file is remote or whether a group is scattered. The return codes are:

0	the file is *not* remote or the group is *not* scattered
1	the file is remote or the group is scattered

Syntax

The format for the $ITSREMOTE function is:

$ITSREMOTE ({[FILE] *name* [AT *location*]

 | [PERM | TEMP] [GROUP] *name*})

where *name* is a %variable or a literal name of the file name, file synonym or group name. You *must* enter the filename in uppercase. If you specify a null for *name*, Model 204 uses the file or group context at the compilation of the statement containing the $function as the default argument.

Example

```
BEGIN
%NAME IS STRING
%NAME = $READ('ENTER THE FILE OR GROUP NAME')
IF $ITSREMOTE(%NAME) THEN
     PRINT %NAME ' IS REMOTE'
ELSE
     PRINT %NAME ' IS NOT REMOTE'
END
```

$JOBCODE

The $JOBCODE function allows a request that is part of one step of a Model 204 batch run to communicate with a subsequent step. $JOBCODE can be used both to examine and to set a completion code by invoking $JOBCODE with the desired completion code as its one argument.

The completion code must fall within the range 0-4095. If the specified completion code exceeds 4095, Model 204 forces the completion code to 4095. No message is output.

Model 204 returns the current step completion code and sets the completion code to the specified value if that value is greater. You can examine the step completion code without changing it by invoking $JOBCODE without an argument or with a null argument.

Using with Online runs

You can include calls to $JOBCODE in the CCAIN stream. In Online sessions, Model 204 maintains two return code values for each user:

- Highest batch return code value received
- Highest Online value

When $JOBCODE is used to set the return code for a user, it sets both values. The value returned from the function, however, is always the batch value. When $JOBCODE was invoked by an Online user in V4R1.1 and earlier, it always returned 0 and did not set the return code.

The $JOBCODE return codes set in an Online run are local to the user who invoked $JOBCODE; they do not affect the return code for other users or for the Model 204 Online.

Example In the following example the first $JOBCODE value, 9, returns a value of 16 because 9 is less that 16, so batch return code is not reset by $JOBCODE(9). The second $JOBCODE value, 21, returns a value of 21 because 21 is greater than 16, so batch return code was reset by $JOBCODE(21).

```
>MSGCTL M204.1030 RETCODEO=8 RETCODEB=16

>@#$%^

 ***  1 M204.1030: INVALID MODEL 204 COMMAND

>B;PRINT $JOBCODE(9);END

 16

>B;PRINT $JOBCODE(21);END

 21
```

See “BATCH2 facility” on page 18-64 for syntax details.

Use to avoid completing jobs

You can use $JOBCODE in conjunction with other functions to avoid completing large jobs that would produce incorrect or unusable results. Suppose a Model 204 run includes a PRINT ALL INFORMATION (PAI) statement that causes all of the fields in a record to be output. A PAI statement sometimes is used to dump the contents of a file before the file is reorganized. After the PAI, new parameters can be specified and the file can be loaded again.

The file manager often does this with the File Load utility (see the *Model 204 File Manager's Guide* for details). If the file being dumped has field level security, it is possible for the PAI run to omit certain fields without issuing error messages. The missing fields are those for which the user does not have READ access. If the output from the PAI run is then reentered to a File Load run, fields are lost. The following technique can be used to avoid this type of problem.

```
OPEN MYFILE
BEGIN
             IF $FLSCHK (,’R’) THEN
                 JUMP TO END.PROCESS
             END IF
             PRINT ’INVALID ACCESS TO FILE’
             PRINT ’STEP COMPLETION CODE WAS’ AND -
                   $JOBCODE (16)
             PRINT ’IT MAY HAVE BEEN SET TO 16’
             STOP
END.PROCESS: IF $JOBCODE () THEN
                 STOP
             END IF
FIND.RECS:   FIND ALL RECORDS
                 .
                 .
                 .
END
```

A File Load step in the same job could be skipped if it were based on a nonzero step completion code. If it is desirable to trigger an ABEND, $JOBCODE can be used in conjunction with the SYSOPT parameter. The 64 (X‘40’) option for the SYSOPT parameter forces an ABEND without a dump at termination, when the step completion code is nonzero.

$LANGSPEC

$LANGSPC returns a string containing the binary value of the specified character in the specified language. You can use $LANGSPC to scan user input for a special character in a language independent manner.

The $LANGSPC syntax is:

$LANGSPC(’*charname*’[,*langname*])

where:

- *charname* is a string containing one of the following values:

  AT
  BACKSLSH
  DOLLAR
  DQUOTE
  EXCLAMAT
  NOT

RBRACE
SHARP
VERTICAL

- *langname* specifies which language to use to obtain the desired code point for the specified character. If the name is not found in NLANG$, the request is cancelled with an error message.

  If a *langname* value of asterisk (*) is specified, the value of the Model 204 LANGUSER parameter determines the language. If no *langname* value is specified, the default is US English (even when the LANGUSER parameter value is not US).

In the following example, the %PATH variable (presumably supplied by the user from the terminal) is searched for the backslash character, regardless of its location in the user terminal's code table:

```
%BACKSLASH IS STRING LEN 1
%BACKSLASH = $LANGSPC('BACKSLSH','*')
%DIR  = $SUBSTR(%PATH, $INDEX(%PATH,%BACKSLASH)+1)
```

$LANGSRT

$LANGSRT translates a given string according to the specified language into a language-neutral binary string against which you can sort.

By determining whether one string is greater or less than another string, you can use $LANGSRT to compare two strings. First apply $LANGSRT to the strings and then compare them using the User Language GT and LT operators.

The $LANGSRT syntax is:

```
$LANGSRT('string'[,langname])
```

where:

- *string* is the original data to be translated into collating sequence.
- *langname* is the name of one of the defined languages, specifying which collating sequence to use. The request is cancelled with an error message if the name is not found in NLANG$.

  If a *langname* value of asterisk (*) is specified, the value of the Model 204 LANGUSER parameter determines the language. If no *langname* value is specified, the default is U.S. English (even when the LANGUSER parameter value is not US).

If the given string contains VARIANT characters, the returned string has encoding information appended. If the encoding information causes the length of the string to exceed 255 characters, only complete byte pairs are appended to the string.

If no complete byte pairs can be appended, nothing is appended (not even the encoding separator) and resulting string lengths can be 253, 254, or 255 bytes.

$LANGUST

$LANGUST translates back to its original form a string previously translated by $LANGSRT. This is useful for applications that maintain sorted arrays of data and need to display the values.

```
$LANGUST('string'[,langname])
```

where:

- *string* is the data in collating sequence to be translated back to its original form.
- *langname* is the name of one of the defined languages, specifying which collating sequence to use. The request is cancelled with an error message if the name is not found in NLANG$.

  If a *langname* value of asterisk (*) is specified, the value of the Model 204 LANGUSER parameter determines the language. If no *langname* value is specified, the default is U.S. English (even when the LANGUSER parameter value is not US).

If encoding information has been truncated, either during $LANGSRT processing or during subsequent expression evaluation, then $LANGUST cannot return the exact string that was originally translated by $LANGSRT.

$LEN

The $LEN function determines the current length of the value of a STRING. The $LEN function takes a fieldname, character literal, or %variable as its one argument.

Example 1

```
$LEN(STATE)
```

equals 4 if the STATE field of the current record contains OHIO.

Example 2

```
$LEN (%NAME)
```

equals 13 if the %NAME variable has a current value of Richard Smith.

$LOWCASE

The $LOWCASE function translates an uppercase or mixed case string into a lowercase string. The translation affects only the letters A–Z. If the first character in the string is alphabetic, the character is converted to uppercase.

Syntax

The format of the $LOWCASE function is:

```
$LOWCASE(string [,language-name])
```

where:

- *string* represents the string to be verified. *string* must be one of:
  - A quoted literal.
  - A %variable.
  - A field name without quotation marks, in which case the current value of the field is verified. In this case, the function call must be embedded in a FOR EACH RECORD loop.
- *language-name* (optional) specifies the language to use. Options are:
  - Omitting this argument, which instructs Model 204 to perform the validation for U.S. English, even if the value of the LANGUSER parameter is not ‘US.’
  - A quoted asterisk (‘*’), which instructs Model 204 to use the value of the LANGUSER parameter to determine which language to use.
  - The quoted literal name of a valid language, for example: NLANGFR1 for French Canadian, Version 1. The request is cancelled with an error message, if the name is not present in NLANG$.

Example

```
$LOWCASE('NAME AND ADDRESS')
```

returns this value:

```
Name and address
```

$LSTFLD

The $LSTFLD function returns field names in alphabetical order and the field description for each field in a file into an image. $LSTFLD requires that you have an image prepared with which Model 204 can create the field description.

Syntax

The format for the $LSTFLD function is:

```
%variable = $LSTFLD([FILE] filename

 [AT location],imagename,loopvar)
```

where:

- *%variable* is the %variable in which the return codes are stored. The return codes are:

Setting	Meaning
0	Success
1	End of field list reached
2	Requested file not available
3	VTBL full or sort not available
4	Error occurred during function processing

- *filename* (required) is the name of the file that contains the fields. A file synonym name can also be used. If you do not enter a location, specifying a null argument, Model 204 uses the reference context (at compile time) of the statement which calls the function. When *filename=groupname* the $LSTDEF function assumes that the name passed is a file name, not a group name.
- *imagename* (required) is the name of the image where the information is to return.
- *loopvar* (required) is the loop variable of the image. You must initialize the loop variable (as in the sample image shown with the $FDEF function) before invoking $LSTFLD. The value returned in *loopvar* is used the next time $LSTFLD is invoked to retrieve subsequent field names.

  Note: Like $LSTPROC, changing any of the loop control information in the $LSTFLD loop variable after the image is initialized is not allowed and might cause the run to snap.

Locating the ZFIELD image

An image is required by the $FDEF and $LSTFLD functions. The ZFIELD image is provided on the Model 204 installation tape. It can give you complete field attribute information; see the $FDEF discussion on page 27-51. The size of the name field for the ZFIELD image is 255 bytes. The location of ZFIELD for your site is listed in this table:

Operating system	Storage location of the ZFIELD image
z/OS	The JCL library
VM	The 2nd tape file (193) as an EXEC
VSE	The JCL library

Example

In this example, the procedure DFIELDS displays the name and definition of each field in the file specified by %FILE.

```
PROCEDURE DFIELDS
BEGIN
*
* include ZFIELD IMAGE as defined for $FDEF*
*
PRINT 'DFIELDS STARTS'
%FILE = '??FILE'
NP
PREPARE ZFIELD
%ZFIELD:BIN1 = 0
%ZFIELD:BIN2 = 0
%ZFIELD:BIN3 = 0
%ZFIELD:BIN4 = 0
%X = 0
REPEAT WHILE %X = 0
   %X = $LSTFLD(%FILE,'ZFIELD',%ZFIELD:LOOPVAR)
   IF %X = 0 THEN
     PRINT %ZFIELD:NAME
     PRINT %ZFIELD:FDEF
   END IF
END REPEAT
PRINT '-- END OF FIELDS RC=' %X
END
END PROCEDURE
```

$LSTPROC

$LSTPROC returns the following procedure information, which is stored in the procedure dictionary:

- Procedure name
- Date and time of last update
- ID of last user to update the procedure
- Length of the procedure (in bytes)
- Procedure security class (if available)
- Procedure file (if in multiple procedure file group context)

Syntax

The format of the $LSTPROC function is:

$LSTPROC(*imagename*, [*procname*][, *loop variable*]

        [, *alias flag*][, *filename*][, *pattern*])

where:

- *imagename* (required) specifies the name of the image into which the information should be returned. The items that must be defined in the image are listed below along with the data type and length of each item. In addition, the position of each item is provided for use with the AT clause where applicable.

Item	Type	Length	Position
Loop variable	BINARY	4	1
Date	PACKED	4	5
Time	STRING	8	9
Length	BINARY	4	17
Security class	BINARY	1	21
Procedure name	STRING	255	22
Procedure file name	STRING	8	
User ID	STRING	10	

  For example:

  ```
  IMAGE PROCS
      LOOPVAR IS BINARY LEN 4
      DATE IS PACKED LEN 4
      TIME IS STRING LEN 8
      LENGTH IS BINARY LEN 4
      CLASS IS BINARY LEN 1
      NAME IS STRING LEN 255
      FILE IS STRING LEN 8
      USERID IS STRING LEN 10
  END IMAGE
  ```

  The procedure file name and the user ID are both optional. If they are not specified, they are not filled in by default values. If the user ID is included, however, the file name must also be included.

  If the image length is not increased, $LSTPROC executes when multiple procedure files have been specified, but does not return file information for each procedure or the user ID.

- *procname* is the procedure for which information should be obtained. The value of *procname* must be the name of a permanent procedure. If this argument is omitted or null, information about the next procedure is returned. You must specify this argument if the alias flag argument has a value of 1.

- *loop variable* is an image item that specifies where to begin the search in the procedure dictionary. Because each invocation of $LSTPROC returns information for only one procedure, this variable is used to control repeated

$LSTPROC calls. The *loop variable* must be the first item in the image. The variable can have one of the following values:

- A null value, if a procedure name is specified.
- 0, for the first iteration of a loop extracting data on all procedures.
- The value of *loop variable* returned by the previous iteration through the loop, if data for the next procedure in the procedure dictionary is being requested.

You must not modify *loop variable* once it is initially set to 0; modifications might result in snaps. A *loop variable* must be specified if one of the following conditions exist:

- A procname argument is not specified.
- The alias flag argument has a value of 1 (aliases is retrieved).

- *flag* (optional) specifies that alias names should be obtained for the specified procedure. The value for the alias flag is 1 to obtain aliases. One alias is returned for each $LSTPROC execution.
- *filename* (optional) specifies the name of the file or group that contains the procedure dictionary. If this argument is omitted, the current file at the time that the request was compiled is used.
- *pattern* (optional) lets you retrieve procedures that match the pattern specified using the standard pattern matching rules beginning on “Pattern matching” on page 4-20.

Status codes

In addition to storing procedure information in an image, $LSTPROC returns a status code that has one of these values:

Status code	Meaning
0	$LSTPROC executed successfully.
1	The end of the procedure dictionary has been reached (there are no more procedure names to process).
2	The specified procedure is not available.
4	An error was encountered during $LSTPROC processing.

Usage notes

- For files created prior to Release 8.1, only the procedure name and security class are provided. An image must be defined earlier in the request in order to receive the information returned by $LSTPROC. For more information on image definition, refer to “Defining an image” on page 17-6.
- $LSTPROC applies to local files only; it is not valid in remote context.

- For a file with a procedure stored in it that was created using Model V3R2.1 or later, then opened under Model 204 V4R1.0 or later with $SYSDATE set to January 1, 2000 (or later), and the procedure is changed, the date returned includes the century--in this example, 100001, in the *cyyddd* format.

Example 1

This request retrieves and prints all procedure names in the current file:

```
BEGIN
    IMAGE PROC
        LOOPVAR IS BINARY LEN 4
        DATE IS PACKED LEN 4
        TIME IS STRING LEN 8
        LENGTH IS BINARY LEN 4
        CLASS IS BINARY LEN 1
        NAME IS STRING LEN 255
        FILE IS STRING LEN 8
        USERID IS STRING LEN 10
    END IMAGE
    PREPARE PROC
    REPEAT WHILE $LSTPROC('PROC',,%PROC:LOOPVAR) = 0
        PRINT %PROC:NAME
    END REPEAT
END
```

Example 2

This request retrieves alias names for a specified procedure:

```
IMAGE PROCINFO
    LOOPVAR IS BINARY LEN 4
    DATE IS PACKED LEN 4
    TIME IS STRING LEN 8
    LENGTH IS BINARY LEN 4
    CLASS IS BINARY LEN 1
    NAME IS STRING LEN 255
    FILE IS STRING LEN 8
    USERID IS STRING LEN 10
END IMAGE

    .
    .
    .
%PROCINFO:LOOPVAR = 0
GETALIAS: IF
$LSTPROC('PROCINFO','MYPROC',%PROCINFO:LOOPVAR,1) =
0
          THEN
PRINT %PROCINFO:NAME
          END IF
   .
```

```
     .
     .
```

Example 3

This request retrieves procedures in a multiple procfile group that match a pattern specified by the user:

```
BEGIN
IMAGE PROCS
  LOOPVAR IS BINARY LEN 4
  DATE IS PACKED LEN 4
  TIME IS STRING LEN 8
  LENGTH IS BINARY LEN 4
  CLASS IS BINARY LEN 1
  NAME IS STRING LEN 255
  FILE IS STRING LEN 8
  USERID IS STRING LEN 10
END IMAGE
%PAT = '??PAT'
NEW PAGE
PREPARE PROCS
PRINT ' PROCEDURE    NAME     DATE    TIME     LENGTH
-
      USERID   CLASS' WITH ' FILE'
REPEAT WHILE $LSTPROC('PROCS',,%PROCS:LOOPVAR) = 0
   IF %PROCS:NAME IS LIKE %PAT THEN
      %DATE =
$DATECNV('YYDDD','MM/DD/YY',%PROCS:DATE)
      PRINT %PROCS:NAME AND %DATE AT 25 AND
%PROCS:TIME -
        AND %PROCS:LENGTH TO 50 AND %PROCS:USERID AT
52 -
         AND %PROCS:CLASS AT 68 -
         AND %PROCS:FILE AT 63
   END IF
END REPEAT
PRINT '-- END OF LIST FOR PATTERN = ' %PAT
END
END PROCEDURE
```

$MISGRUP

The $MISGRUP function is used with the ON units ON MISSING FILE and ON MISSING MEMBER. It returns the group name if the error occurred in group context, or null if the error occurred in file context.

Example

The following is an example of an ON MISSING FILE unit that uses the $MISGRUP, $MISLOC, $MISNAME, and $MISNUM functions:

```
ON MISSING FILE
   %X = $MISGRUP
   IF %X = ’’ THEN
      * THIS IS FILE CONTEXT BECAUSE $MISGRUP
RETURNED NULLS
      PRINT ’MISSING FILE’ AND $MISNAME AND ’AT’ AND
$MISLOC
   ELSE
      PRINT ‘MISSING GROUP' AND %X AND 'FILES
FOLLOW:'
      FOR %I FROM 1 TO $MISNUM BY 1
         PRINT $MISNAME(%I) AT 13 AND 'AT' AND
$MISLOC(%I)
      END FOR
   END IF
END ON
```

$MISLOC

The $MISLOC function is used with the ON units ON MISSING FILE and ON MISSING MEMBER. It returns the location of a missing member or file. See the example given for “$MISGRUP” on page 27-77.

$MISNAME

The $MISNAME function is used use with the ON units ON MISSING FILE and ON MISSING MEMBER. It returns the file name of a missing member or file. See the example given for “$MISGRUP” on page 27-77.

$MISNUM

The $MISNUM function is used with the ON units ON MISSING FILE and ON MISSING MEMBER. It returns the number of files that failed in the group. See the example given for “$MISGRUP” on page 27-77.

$MISSTMT

The $MISSTMT function is used with the ON units ON MISSING FILE and ON MISSING MEMBER. It returns the statement that caused the ON unit to be entered.

The possible return values (statement names) of $MISSTMT are:

```
ADD
BACKOUT
CHANGE
CLEAR LIST
COMMIT
COUNT
DELETE ALL RECORDS
DELETE EACH
DELETE FIELD
DELETE RECORD
FIND ALL VALUES
FILE
FIND
FOR
FOR EACH VALUE
FOR RECORD NUMBER
INSERT
PLACE RECORD
PLACE RECORDS
RELEASE RECORDS
RELEASE ALL RECORDS
REMOVE RECORD
REMOVE RECORDS
SORT
SORT VALUES
```

$MOD

The $MOD function returns the remainder that results when the first argument is divided by the second argument (the first argument modulo the second argument). Each argument is first rounded to an integer. $MOD (X,0) is defined to be X.

Example 1

```
$MOD (5,3)
```

equals 2.

Example 2

```
IF $MOD (Z,2) THEN PRINT ’ODD’
```

prints ODD only if the value of the field Z is odd.

$OCCURS

The $OCCURS function interprets a character string argument as a field name in the following manner:

- In file context, $OCCURS returns 0 if the field description in the current file containing the specified field does not include the OCCURS field attribute. $OCCURS returns a number representing the number of specified occurrences if the field has OCCURS in its description.
- In group context, $OCCURS returns 0 only when the field is described without an OCCURS attribute in every file making up the group. Otherwise, the function returns the minimum OCCURS specification included in all of the files.

- If the field specified as the $OCCURS argument is not defined in the current file or group, $OCCURS returns a value of -1.

Example

```
$OCCURS ('AGENT')
```

equals 1 if the description of AGENT contains OCCURS 1.

$ONEOF

The $ONEOF function is a table lookup function that can replace a series of IF conditions.

Syntax

```
$ONEOF('item','set',delimiter')
```

where:

- *item* is an element that might be in *set*.
- *set* is one or more items
- *delimiter* is the character used to separate the set items.

Example

```
$ONEOF('MALE', 'FEMALE/MALE','/') equals 1
$ONEOF('GRAY','BLACK*WHITE*GRAY','*') equals 1
```

- If the third argument is omitted, a semicolon is assumed to be the delimiter used in the second argument. For example:

  ```
  $ONEOF('FEMALE', 'FEMALE;MALE') equals 1
  ```

- If the second argument is null, or if the first argument is not an element of the second argument, 0 (false) is returned. For example:

  ```
  $ONEOF('BOY','FEMALE;MALE') equals 0
  $ONEOF(",'FEMALE;MALE') equals 0
  $ONEOF('BOY',") equals 0
  ```

- You can enter null values as the following examples show.

  ```
  $ONEOF(",';FEMALE;MALE') equals 1
  $ONEOF(",'FEMALE;;MALE') equals 1
  $ONEOF(",'FEMALE;MALE;') equals 1
  ```

- If a long string of values is used frequently, a special record containing those values can be useful. In the following example, a record is created with a field consisting of a list of departments. The list is used in the second request to locate and process records for which the department is not known.

```
BEGIN
DECLARE %DEPT STRING LEN 50
            STORE RECORD
                TABLE = VALS
                DEPT VALS = DEPT1/DEPT2/DEPT3/etc.
            END STORE
END
            .
            .
            .
BEGIN
FIND.RECS: FIND ALL RECORDS FOR WHICH
               TABLE = VALS
           END FIND
           FOR EACH RECORD IN FIND.RECS
               %DEPT = DEPT VALS
           END FOR
PAY.RECS:  FIND ALL RECORDS FOR WHICH
              TYPE = PAYROLL
           END FIND
           FOR EACH RECORD IN PAY.RECS
               IF NOT $ONEOF(DEPT,%DEPT,'/') THEN
                   .
                   .
                   .
```

$PACK

The $PACK function returns the packed decimal representation of the character string. $PACK is used as an item in an assignment to a %variable that is written to a sequential USE dataset. $PACK should not be used as an operand in an arithmetic expression, because the value returned cannot be correctly interpreted as a number.

Syntax

The format for the $PACK function is:

$PACK(*value,precision* [,*scale*][,'UNSIGNED'])

where:

- *value* is a character string that can be a field name, a %variable, or any type of string expression.
- *precision* is the number of digits required in the number to be returned. You must specify the precision.

- *scale* specifies the number of digits to be returned to the right of the decimal point. The scale argument is optional. If the scale argument is omitted, the returned result is an integer.
- *UNSIGNED* specifies that the packed decimal data has a sign code of X‘F’ regardless of the sign of the data value. The UNSIGNED argument is optional. If the UNSIGNED argument is omitted, the data contains the appropriate code for the sign of the data value.

Model 204 pads the result with leading integer or trailing fractional zeros as appropriate. If the integer portion of the value string is too long, or if the value is nonnumeric, $PACK returns binary ones. Model 204 interprets the value returned by $PACK as a character string.

Example

If the user were to specify these statements:

```
%X = 123.4
%A = $PACK(%X,8,3,’UNSIGNED’)
```

the result in %A would be X‘0123400F’. The result would be identical if %X were assigned a value of -123.4.

$PAD

The $PAD function allows padding to the left with a designated character. The $PAD function takes three arguments and returns a character string of the length specified by the value of the third argument. The resulting string contains the first argument, right-justified, and padded with the character indicated in the second argument. The length argument is rounded if it is not an integral value. For example:

```
$PAD(’55449825’,’0’,9) equals ’055449825’
$PAD(’123.65’,’*’,8) equals ’**123.65’
$PAD(’123.65’,’*’,7.66) equals ’**123.65’
```

- If the value of the length argument is greater than 255, 255 is used.
- If the value of the length argument is not greater than zero, a null string is returned.
- If the second argument contains more than one character, the first character in the string is the one used for padding. For example:

  ```
  $PAD(’123.65’,’*4’,8) equals ’**123.65’
  ```

$PADR

The $PADR function allows padding to the right with a designated character. $PADR works exactly like $PAD with the exception that $PAD pads to the left, while $PADR pads to the right. For example:

```
$PADR('123.65','*',9) equals '123.65***'
```

$POST

Function

Indicates that some event has occurred. The thread(s) waiting on the Event Control Block (ECB) can resume processing.

Syntax

```
$POST({ECB-number[,post-code][,string] | 'QZSIG'})
```

Where:

- *ECB-number* is a string with a numeric value from one to the NECBS parameter that identifies the ECB to be posted. The *ECB-number* can be expressed as a numeric literal, a %variable, or a field name.
- *post-code* is a numeric value between zero and 16,777,215. You can use this optional argument for whatever you wish. If omitted, a default post code of zero is used.

  The post code is accessible using the $ECBTEST function followed by the $STATUSD function for the specified posted ECB.

  Once set, unless explicitly reset to zero, post codes persist whether or not the ECB is posted.
- *string* can be up to 255 bytes long. It can be a numeric, a literal enclosed in quotation marks, a %variable, or a field name.

  String data is accessible using the $ECBDGET function for the specified ECB.

  Once set, unless explicitly reset to null, data strings persist whether or not the ECB is posted. Depending on the sequence, data strings can be changed by either the $POST or $ECBDSET functions.
- QZSIG is an extended quiesce named ECB. To use it, you need not set the NECBS parameter. QZSIG can be expressed as a literal, a %variable, or a field name. Although QZSIG can be posted, it cannot be unposted. Unposting is handled internally at the end of the extended quiesce. You cannot specify a post code or a string for QZSIG.

Usage

The $POST function posts a specified ECB with an optional post code and an optional associated string. Only an unposted ECB may be posted. If an ECB is already posted, the posted state is indicated by return code 14. The ECB contents and associated string do not change. Any other thread waiting ($WAIT) on the specified ECB becomes eligible to resume processing.

Use the $POST, $UNPOST, and $WAIT functions to coordinate processing between threads.

The following return codes apply to the $POST function

Return code	Meaning
0	Success
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The first argument is less than one or greater than the NECBS parameter
5	NUSERS = 1
8	No argument specified
9	Checkpointing inactive
10	Post code out of range, 0 to 16,777,215
11	The QZSIG ECB is already posted
12	Invalid argument CPQZ specified or invalid argument following QZSIG
13	Cannot be issued outside of extended quiesce
14	An attempt to post an already posted ECB

Example

On Thread A

```
BEGIN
PRINT $UNPOST(1)
PRINT $WAIT(1)
PRINT $STATUSD
END
```

On Thread B

```
BEGIN
PRINT $POST(1,32)
END
```

In the previous code, the $UNPOST function on Thread A marks the ECB specified as unposted. Then, Thread A waits at the $WAIT(1) until another thread issues a $POST(1,*nnnn*). Thread B posts the ECB number one with $POST(1,32), at which point Thread A resumes evaluation with PRINT $STATUSD and prints the post code of 32.

$RDPROC

The $RDPROC function sequentially retrieves the lines of a User Language procedure that is stored in a Model 204 file. The lines of the User Language procedure can reside in a file or in a group. A group can contain a single procedure file or multiple procedure files. Multiple procedure files are searched in the order in which they appear in the DEFINE GROUP command.

You can also read temporary procedures by specifying the procedure number, for example, 0, -1, or -2, in place of the procedure name. An input file is not required.

$RDPROC applies to local files only; it is not valid in remote context.

Options

The four options for $RDPROC are:

Option	Action
OPEN	Locates and locks a procedure
GET	Reads a line from a procedure
CLOSE	Explicitly unlocks the procedure at any time
LINEND	Reads a line end character from a procedure

Single request

You must open and read a procedure within one request, without leaving the request. If the procedure is a permanent procedure, $RDPROC holds a share lock on the procedure while reading it, which prevents other users from modifying the procedure. $RDPROC unlocks the procedure automatically when it reads end-of-procedure.

The $RDPROC function uses 336 bytes of STBL and 8 bytes per $RDPROC level. You can nest $RDPROC up to the value of the MAXINCL parameter, plus one.

Syntax for locating and locking a procedure

The format for the $RDPROC function to locate and lock a procedure is:

```
ctlid = $RDPROC('OPEN', fgname, procname)
```

where:

- *ctlid* is the integer ID of an internally maintained control field that is passed into $RDPROC for GET, LINEND, and CLOSE processing.
- *fgname* is a name of the file or group that contains the procedure. The format of fgname is:

```
[[[TEMP | PERM] GROUP] | [FILE]] identifier
```

where *identifier* is the name of the group or file.

- *procname* is the name of a procedure.

Note: With $RDPROC, you can open up to six procedures concurrently.

Syntax for reading a line from a procedure

The format for the $RDPROC function to read a line from a procedure is:

```
text = $RDPROC('GET',ctlid)
```

where:

- *text* is the next line of the procedure (null if end-of-procedure is reached).
- *ctlid* is the integer ID of an internally maintained control ID that is assigned during OPEN processing.

Syntax for closing and unlocking a procedure

The format for the $RDPROC function to close and unlock a procedure is:

```
text = $RDPROC('CLOSE',ctlid)
```

where:

- *text* is set to the null string.
- *ctlid* is the integer ID of an internally maintained control ID that is assigned during OPEN processing.

Syntax for reading a line end from a procedure

The format for the $RDPROC function to read a line-end character from a procedure is:

```
text = $RDPROC('LINEND',ctlid)
```

where:

- *text* is the value of the line-end character parameter in effect when the procedure was saved. By default, this is a semicolon (;).
- *ctlid* is the integer ID of an internally maintained control ID that is assigned during OPEN processing.

How $RDPROC works

Possible return status settings are summarized in this table:

Setting	Meaning
0	Normal completion (a procedure line was read; more lines follow)
1	End-of-procedure reached (no more procedure lines exist)
2	Error occurred (see $STATUSD settings)

If the completion status is 2 and indicates that an error occurred or that the EOP was read, the procedure is automatically closed. A subsequent $RDPROC CLOSE call is unnecessary and returns this message:

`$STATUS=2 / $STATUSD=11 (`*`invalid ctlid value`*`)`

$STATUSD setting

The STATUSD setting can be one of the settings summarized in the following table:

Setting	Meaning
1	Argument 1 is missing or null.
2	Argument 2 is missing or null.
3	Argument 3 is missing or null
4	Argument 1 must be OPEN, GET, CLOSE, or LINEND.
5	Invalid context specification in argument.
6	GROUP has no procedure file.
7	Could not lock on specified procedure.
8	Could not find procedure or access not allowed.
9	(unassigned)
10	Maximum number of procedures (five) are already open.
11	Ctlid contains an invalid value.

Example

```
BEGIN
DECLARE %PROC   IS STRING LEN 50
* USE LEN 19 BELOW TO ALLOW FOR 'TEMP GROUP XXXXXXXX'
DECLARE %FILE   IS STRING LEN 19
DECLARE %TEXT   IS STRING LEN 255
DECLARE %CTLID  IS FIXED
*
```

```
* FILE MUST BE PREVIOUSLY OPENED
*
            %FILE = $READ('FILE/GROUP CONTEXT?')
            %PROC = $READ('PROCEDURE NAME?')
*
* OPEN THE PROCEDURE
*
            %CTLID = $RDPROC ('OPEN', %FILE, %PROC)
            IF $STATUS > 0 THEN
               PRINT '$RDPROC OPEN ERROR, REASON CODE=' -
               AND $STATUSD
               STOP
            END IF
*
* DETERMINE THE LINEND CHARACTER
*
            %TEXT = $RDPROC ('LINEND', %CTLID)
            IF $STATUS > 0 THEN
              PRINT '$RDPROC LINEND ERROR, REASON CODE=' -
              AND $STATUSD
              STOP
            END IF
            PRINT 'LINEND CHARACTER = ' WITH %TEXT
            PRINT
*
* DISPLAY THE PROCEDURE
*
            REPEAT WHILE $STATUS = 0
            %TEXT = $RDPROC ('GET', %CTLID)
            PRINT %TEXT
            END REPEAT
*
* SEE IF WE ENDED ABNORMALLY
*
            IF $STATUS > 1 THEN
              PRINT 'GET ERROR, REASON CODE=' AND $STATUSD
            END IF
*
* CLOSE THE PROC (UNNECESSARY SINCE WE READ TO EOP)
*
            %TEXT = $RDPROC ('CLOSE', %CTLID)
END
```

Opening an empty temporary procedure

If you attempt to open an empty temporary procedure, $STATUS returns a zero and issues the following error:

```
M204.1172 PREVIOUS REQUEST NOT DEFINED
```

The current request continues as normal.

$RDPROC and temporary procedures

Using $RDPROC, create a temporary procedure named -1 and print procedure -1 to procedure -2. In this example, the place for the input file name is blank (, ,) or NULL.

```
USE PROC -2
MONITOR
BEGIN
        %CTLID = $RDPROC('OPEN',,-1)
         IF $STATUS THEN
             JUMP TO BAILOUT
         END IF
         REPEAT WHILE NOT $STATUS
             PRINT   $RDPROC('GET',%CTLID)
         END REPEAT
BAILOUT:
END
```

$READ

The $READ function enables the user to enter data from the terminal as a request is evaluated. $READ is useful when creating generalized requests. You can also use $READ to read a sequential input file; see “Using the $READ function to read sequential input” on page 27-92.

Syntax

The format for the $READ function is:

$READ([*prompt*][,*option*])

where:

- *prompt* is the literal to appear on the terminal, prompting for input. This argument is optional.
- *option* indicates how the input value is to be returned. The only valid option is ‘TRANSPARENT’. When this argument is present, $READ does not translate LINEND characters as the logical end of line nor a hyphen as the continuation character, remove trailing blanks, or convert lowercase characters to uppercase. $READ simply returns the input line of up to 255 characters exactly as entered.

Prompting the user for a line of input

Each time $READ is evaluated, Model 204 prompts the user for a line of input. The response becomes the character string value returned by the function. For example:

```
$READ('HELLO')
```

causes Model 204 to respond:

```
$$HELLO
```

and wait for the user to enter a reply each time the function is evaluated.

If the user keys in:

```
GOODBYE
```

in reply, the function returns a value equal to the quoted string ‘GOODBYE’.

If no prompt argument is specified, as $READ (), the prompt is:

```
$$
```

The end of the user’s reply is delimited by a carriage return or a semicolon. However, if the user ends the first line of reply with a hyphen, the carriage return is ignored, and the reply can be continued on another line.

Including $READ functions and dummy strings in a procedure

If $READ functions are included in a procedure, one or more responses to the $READs can be specified in the command that invokes the procedure, along with responses to dummy strings included in the procedure. The following procedure contains a typical data entry request.

```
PROCEDURE ENTRY
BEGIN
NAME.AND.AGE: %A = $READ('NAME')
              %B = $READ('AGE')
              STORE RECORD
                  TYPE = ??TYPE
                  NAME = %A
                  AGE = %B
              END STORE
              IF $READ('MORE?') EQ 'YES' THEN -
                  JUMP TO NAME.AND.AGE
              END IF
END
END PROCEDURE
```

You can include the procedure and specify responses for the $READ functions and dummy strings included in the procedure by issuing the following:

```
INCLUDE ENTRY, PERSONNEL, ROBERT, 27, YES, LOUISE,
28, NO
```

Dummy string responses must appear first, followed by $READ responses. If any $READ responses are included in the list, it also must include a response for each dummy string encountered before the first $READ is executed.

As dummy strings and $READs are encountered in the procedure being executed, entries are retrieved from the list specified in the INCLUDE or IF line. No prompts are issued for responses taken from the list. If there are more dummy strings and $READs than responses in the list, Model 204 prompts for a response as if no list had been specified.

Alternatively, if the user responds to a request with several $READs in a row, the user can wait for the first prompt and then enter several replies at once, separating them by semicolons. Model 204 prints the prompts for the next $READs but does not wait for the user to reply to each one.

Results of $READ prompts

Results of $READ prompts contain leading or intermediate blanks that the user enters if the TRANSPARENT argument is not specified. However, trailing blanks are dropped. This occurs when the ENTRY procedure shown earlier is included. Note the stacking of responses on a single input line.

```
INCLUDE ENTRY
??TYPE
PERSONNEL
$$NAME
ROBERT
$$AGE
27
$$MORE?
YES
$$NAME
LOUISE 28 28
$$AGE
$$MORE?
YES
$$NAME
DALE 24; YES; RICHARD; 37; YES: THO-
MAS 59; NO
$$AGE
$$MORE?
$$NAME
$$AGE
$$MORE?
$$NAME
$$AGE
$$MORE?
```

This example illustrates the differences between dummy strings and the $READ function.

Dummy strings allow the user to fill in pieces of the request text just before compilation begins. The user’s replies to ?? prompts are substituted before the text is compiled. ??TYPE is replaced only once, at compilation time, by PERSONNEL.

$$ prompts and $READ function input not specified at INCLUDE time are processed when the request is evaluated. Substitutions for these prompts are accepted as character strings; no text or character evaluation is made. Consider the following example:

```
BEGIN
PRINT.COUNT: FIND AND PRINT COUNT
                 AGE = ??AGE
             END FIND
END

BEGIN
             %AGE = $READ('ENTER AGE')
PRINT.COUNT: FIND AND PRINT COUNT
                 AGE = %AGE
             END FIND
END
```

In the first request, if the user replies “10 OR 11” to the ??AGE prompt, the line AGE = 10 OR 11 is properly compiled and the request searches for records with AGE either 10 or 11. However, if the user replies 10 OR 11 to the $$ENTER AGE prompt in the second request, 10 OR 11 is considered a character string and Model 204 searches for records with AGE fields containing the string ‘10 OR 11’.

If the user presses the attention key (ATTN, BREAK, or PA1, depending on terminal type) or enters **`*CANCEL`** in response to a $READ prompt, Model 204 performs the action specified in the ON ATTENTION unit (see “ON units” on page 12-21) if one is specified in the request.

If ON ATTENTION is not specified, Model 204 terminates the request and all nested procedures. Control is returned to the command level at the user’s terminal.

Using the $READ function to read sequential input

You can use the $READ function to read sequential input:

1. Run Model 204 in batch mode. $READ assumes terminal input in online mode and sequential input (for example, tape or disk) in batch mode.
2. Concatenate a User Language procedure to the sequential input to be read. The following JCL is required:

   ```
   // CCAIN DD DSN=USER.LANG.PROCEDURE,DISP=OLD
            DD DSN=SEQU.INPUT.FILE,DISP=OLD
   ```

3. Use the $SUBSTR function to parse the input. For example:

   ```
   %FIELDA=$SUBSTR(%INPUT,1,10)
   %FIELDB=$SUBSTR(%INPUT,11,5)
   ```

The input is read into the %variable, for example, INPUT). It cannot exceed 255 characters in length.

$READINV

The $READINV function is identical to the $READ function, except that:

- Input from the terminal is not echoed. It is treated the same as a password for the device would be treated.
- $READINV cannot take its response from arguments saved from the INCLUDE line.

The $READINV function takes a field name as its one argument.

$READLC

The $READLC function is identical to $READ, except that it deactivates case translation, regardless of the current *UPPER/*LOWER setting. $READLC takes one argument.

$REMOTE

The $REMOTE function returns the SNA Communications Server (formerly VTAM) network ID (SNA Communications Server applid, or user zero parameter VTAMNAME) of the Model 204 region from which a user is transferred by the Transfer Control facility. This SNA Communications Server applid should be the DEFINE PROCESSGROUP command REMOTEID value of the Model 204 region to which the user is transferred. $REMOTE simplifies a return transfer to the region from which the user transferred: the $REMOTE value is used as the target of the return transfer.

It returns an 8-byte character string consisting of the SNA Communications Server applid of a remote Model 204 region. If a user transferred into the Model 204 region from a non-Model 204 region, $REMOTE returns blanks. $REMOTE can only be issued from an IODEV=7 thread. The $REMOTE function takes no arguments.

$REVERSE

The $REVERSE function reverses the order of a string.

Syntax

The format of the $REVERSE function is:

```
$REVERSE(string)
```

where *string* is any character string.

For example:

```
PRINT $REVERSE('AB CDEF')
```

prints the value FEDC BA.

$REVERSE is particularly useful when the file manager defines an ORDERED field to contain the characters of a frequently retrieved field in reverse order.

Thus, when leading wildcard patterns (for example, *SON) are used to retrieve records or values, the patterns can be reversed to optimize the ordered retrieval. If the patterns remain in leading wildcard form and the original, unreversed field is used to retrieve values or records, the entire Ordered Index is searched to find the values satisfying the pattern.

For example, an ORDERED field named WORD exists and values ending in LY are to be retrieved frequently. To optimize this retrieval, another ORDERED field named REVERSE.WORD can be defined to store the reversed values in the WORD field, and the pattern *LY can be reversed to retrieve values in REVERSE.WORD LIKE ‘YL*’. The pattern YL* optimizes the Ordered Index retrieval, providing much faster performance than *LY.

For more information on defining fields to optimize leading wildcard patterns, refer to the *Model 204 File Manager’s Guide*.

$RLCFILE

The $RLCFILE function returns the name of the file in which the last record locking conflict occurred. If the file is remote, $RLCFILE also returns the location of the file (in the form `filename AT location`). This function is most useful when used within an ON FIND CONFLICT or ON RECORD LOCKING CONFLICT unit.

The $RLCFILE function takes no arguments. At the beginning of a request, $RLCFILE is set to null values. After a record locking conflict occurs, $RLCFILE returns the name of the conflicting file.

Example

This request prints the name of the file that invokes the ON RECORD LOCKING CONFLICT unit:

```
BEGIN
ON RECORD LOCKING CONFLICT
    PRINT ’UNABLE TO COMPLETE’
    PRINT ’CONFLICT OCCURING IN FILE = ’ WITH
$RLCFILE
END ON

GET.NAME: FOR EACH RECORD WHERE -
              FULLNAME IS LIKE ’JOHNST*’
              CHANGE AGENT TO GOODRICH
          END FOR
END
```

$RLCREC

The $RLCREC function returns the internal record number for which the last record locking conflict occurred. This function is most useful when used within an ON FIND CONFLICT or ON RECORD LOCKING CONFLICT unit.

At the beginning of a request, $RLCREC is set to -1. After a record locking conflict occurs, $RLCREC returns the internal record number of the record in conflict. The $RLCREC function takes no arguments.

Example

This request prints the internal record number of the record that invokes the ON RECORD LOCKING CONFLICT unit:

```
BEGIN
ON RECORD LOCKING CONFLICT
    PRINT 'UNABLE TO COMPLETE'
    PRINT 'CONFLICTING WITH RECORD # ' WITH $RLCREC
END ON

GET.NAME: FOR EACH RECORD WHERE -
              (FULLNAME IS GREATER THAN COLLUM
             POLICY NO IS BETWEEN 100340 AND 100492)
              CHANGE STATE TO CALIFORNIA
          END FOR
END
```

$RLCUID

The $RLCUID function returns the user ID that has caused an ON FIND CONFLICT or ON RECORD LOCKING CONFLICT to occur. $RLCUID takes no arguments and returns a variable length character string. If the conflicting user is on a remote system the return value is in the form:

userid AT *location*

$RLCUSR

The $RLCUSR function returns the user number of the user with which the request conflicted when the last record locking conflict occurred. This function is most useful when used within an ON FIND CONFLICT or ON RECORD LOCKING CONFLICT unit.

At the beginning of a request, $RLCUSR is set to null values. After a record locking conflict occurs, $RLCUSR returns the user number of the conflicting user. The $RLCUSR function takes no arguments.

Example

This request prints the user number of the user that invokes the ON RECORD LOCKING CONFLICT unit:

```
BEGIN
ON RECORD LOCKING CONFLICT
    PRINT 'UNABLE TO COMPLETE'
    PRINT 'CONFLICTING WITH USER # ' WITH $RLCUSR
```

```
END ON

GET.NAME: FOR EACH RECORD WITH -
              FULLNAME LIKE 'BAKER*,DEROUCHE,TANGO*'
              CHANGE AGENT TO CASOLA
          END FOR
END
```

$ROUND

This function rounds a number to a specified number of decimal places.

Syntax

The format of the $ROUND function is:

$ROUND(*number*, *places*)

where:

- *number* is the number to be rounded.
- *places* is the number of significant decimal places to which the number should be rounded.

$ROUND returns the first argument, rounded to the number of decimal places specified in the second argument. If the second argument is negative, $ROUND returns the first argument unaltered and prints an error message. Omitted arguments are set to zero.

Example

```
$ROUND(200.565, 2) equals 200.57
$ROUND(200.565, 0) equals 201
$ROUND(200.565) equals 201
$ROUND(-200.565, 1) equals -200.6
$ROUND(-200.565, 2) equals -200.57
```

$SCAN

$SCAN is an alias for $INDEX.

$SCLASS

$SCLASS returns a variable length character string equal to the current user's subsystem user class (SCLASS). $SCLASS returns a null string if the user is not running in a subsystem. For more information on user classes and subsystems, refer to Chapter 23.

The $SCLASS function typically is used when designing applications through the Subsystem Management facility. $SCLASS can be used to determine the user class of the current user. Control can then be transferred depending upon the user’s privileges. The $SCLASS function takes no arguments.

Example

```
GET.OPTION: JUMP TO (ADD.REC,VIEW.REC,UPD.REC) -
                 %MAIN.MENU:SELECTION
                    .
                    .
                    .
UPD.REC:    IF $SCLASS = ’UPDATE’ THEN
               IF $SETG(’NEXT’,’PRE-MAINT.PGM’) THEN
                    PRINT ’GLOBAL TABLE FULL’
                END IF
                    .
                    .
                    .
```

$SETG

The $SETG function performs two tasks. It attempts to create or change an entry in the global variable table and also informs the user if the operation was successful. $SETG returns a 1 (true) if the global variable was not stored due to lack of space. It returns a 0 (false) if the variable was successfully stored.

$SETG takes two arguments. The first argument contains the name of the global variable; the second argument contains the value. Previously stored variables with the same name are deleted first.

Example

The following statement attempts to store a global variable with a name of GLOB and with a value equal to the character string returned from the $READ. A message is to be printed if the operation was not successful.

```
IF $SETG(’GLOB’,$READ(’ENTER GLOBAL VALUE’)) THEN
    PRINT ’HELP’
END IF
```

Refer to Chapter 20 for a detailed explanation of global variables and examples of the $SETG function within a request.

$SETL

The $SETL function sets the current line counter for the output device currently in effect to the value specified as the $SETL argument. Use the $SETL function only when routing output to an external dataset. You cannot use $SETL with full-screen devices.

$SETL returns a number representing the maximum physical line length allowed on the current output device. This line length is determined from the

OUTMRL and OUTCCC parameters or, if an alternate output device is being used, from the LRECL specification on the USE dataset DD statement and the UDDCCC parameter. All parameters are described in the *Model 204 Command Reference Manual*.

The $SETL function does not ordinarily reposition the output device or perform any input/output operations. It simply alters the value of the line counter for the device. When this counter is compared to the value of OUTLPP or UDDLPP, the effective size is changed for the current page. If the counter is set to a value equal to or greater than the effective lines-per-page value (OUTLPP or UDDLPP), a NEW PAGE action is forced.

$SETP

This function sets the current page number for the output device currently in effect. The current page number is used by Model 204 in formatting HEADER 0 or filling in a value for the $HPAGE function. The value specified as the $SETP argument becomes the current page number. Note, however, that the current page number is incremented *before* the next page is printed, so if you want the next page to be page 5, use 4 as the argument to $SETP as shown in the following example:

```
%PAGENO = $SETP(4)
SET HEADER1 WITH $HPAGE(%PAGENO)
```

The first line in the example is sufficient for setting the page number in HEADER 0. If you want the page number to appear in any other header line, you must do it indirectly (using the $HPAGE function), as in the second line of the example. The $SETP function is not valid in a SET HEADER statement.

Although the effect of $SETP is to set the current page number, its return value is a number representing the current number of output lines per page (the value of the OUTLPP parameter or, if an alternate output device is being used, the UDDLPP parameter). This is the value you would see if you printed the value of %PAGENO in the example (*not* the current page number).

$SLSTATS

The $SLSTATS function lets you evaluate the relative expense of different processes in a single request by resetting the recording of since-last statistics at the point in a request where the function call appears.

Syntax

The format of the $SLSTATS function is:

```
$SLSTATS([’string’ | %variable])
```

Model 204 uses the first four characters in the quoted string, or in the current value of the %variable, as a label to identify the section of the request to which the corresponding since-last audit trail entry applies.

The default value for the argument is EVAL. Because the recording of since-last statistics begins automatically with request execution, the first set of since-last statistics in the audit trail for any request is always be labelled EVAL. This is true even if the first statement in a request contains a $SLSTATS call.

You can call the $SLSTATS function by embedding it in a PRINT statement or assigning it to a %variable. For example, the following statement initiates a new set of since-last statistics (labelled FND1) for the remainder of the request, or until the next $SLSTATS call appears:

```
%X = $SLSTATS('FND1')
```

Note that each invocation of $SLSTATS produces a new SMF (System Monitoring Facility) record. If the identifying label is specified, then it replaces EVAL in the SMF record, as it does in the audit trail.

In addition to EVAL, there are several other labels generated by Model 204 for audit trail and SMF records. These include:

```
CMPL
COPY
DUMP
EDIT
LOAD
REST
```

You can assume any label not listed above to be generated by an $SLSTATS call. (Note, however, that there is no prohibition against using the above labels with $SLSTATS.) For accounting purposes, any label generated by $SLSTATS should be included as EVAL statistics.

See the *Model 204 System Manager's Guide* for detailed information on since-last statistics.

$SNDX

The $SNDX function returns the SOUNDEX code of an argument. The $SNDX function is commonly used with files containing unusual or frequently misspelled names. You can create a field containing the SOUNDEX code for a name and then use that field for retrievals.

The code is derived in the following manner:

1. All consecutive occurrences of the same letter are reduced to a single occurrence.

2. The first character of the string becomes the first character of the result.

3. All vowels, special characters, and the letters H, W, and Y are eliminated.

The rest of the characters are transformed as follows:

Character(s)	**Change to…**
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

4. SOUNDEX code assigns numbers to the next three consonants of the word following the number assignments shown in the previous table, but disregards any remaining consonants. The Model 204 $SNDX function continues to assign numbers to all consonants.

Example 1

```
$SNDX('MURRAY') equals M6
$SNDX('MARY') equals M6
$SNDX('O"MALLEY') equals O54
```

Example 2

```
BEGIN
           %A = $SNDX($READ('ENTER NAME'))
FIND.RECS: FIND ALL RECORDS FOR WHICH
               NAME SOUND = %A
           END FIND
.
.
.
```

$SQUARE

The $SQUARE function multiplies a number by itself. $SQUARE takes the number to be squared as its one argument. For example:

```
$SQUARE(8)
```

equals 64.

$STAT

The $STAT function returns the current value of any user final (LOGOUT) or partial statistic. This function is useful for determining which resources are used by various portions of a request.

Syntax

The format of the $STAT function is:

```
$STAT(statistic,'user')
```

where:

- *statistic* specifies the name of the user statistic to be returned. Refer to the *Model 204 System Manager's Guide* for a complete list of user statistic names and meanings.
- *user* specifies the return of a user final or partial statistic.

Example

```
     .
     .
     .
%X = $STAT('CPU','USER')
     .
     .
     .
```

$STATUS

The $STATUS function returns a numeric value that indicates the success or failure of the last executed OPEN/OPENC statement, external I/O statement or program communication statement.

For a discussion of…	See
Error handling with the OPEN and OPENC statements	"Error handling" on page 16-16
External I/O statements	"Reading external files or terminal input" on page 17-4
Communication statements	Chapter 18

- List of $STATUS return values, refer to Table 18-2 on page 18-10. The $STATUS function takes no arguments.

Example

```
     .
     .
     .
READ IMAGE ACCT.RECV.REC
    IF $STATUS = 0 THEN
        CALL PROCESS.AR
    ELSE
```

```
        LOOPEND
    END IF
     .
     .
     .
```

$STATUSD

The $STATUSD function returns a numeric value that indicates a more detailed description of a condition reported by $STATUS. The $STATUSD function is valid only for program communication statements. For more information on:

- Communication statements, refer to Chapter 18
- A list of values that $STATUSD returns, refer to Table 18-12 on page 18-72. $STATUSD takes no arguments.

Example

```
    .
    .
    .
SEND %CMSPROGRAM TO %CMSRECEIVE
IF $STATUS EQ 4
    IF $STATUSD EQ 1669 THEN
        READ IMAGE EMP.REC FROM VSAMDS1
    ELSE
    .
    .
    .
```

$STATUSR

Function

Resets the value of $STATUS to zero and returns a value of zero.

Syntax

```
$STATUSR
```

Usage

The $STATUSR function accepts no arguments.

Programs that use a DO WHILE $STATUS=0 loop can use $STATUSR within the loop to process a nonzero $STATUS and continue looping.

$STRIP

The $STRIP function returns the contents of an argument with leading zeros suppressed. $STRIP, like $PAD, is useful for report formatting.

Example

```
$STRIP('055449825') equals '55449825'
$STRIP('00000') equals ''
```

If a character other than zero is in the first position of the argument, zeros within the string are not removed. For example:

```
$STRIP(' 055449825') equals ' 055449825'
```

$SUBSTR

The $SUBSTR function returns a substring of a string. $SUBSTR is identical to $DEBLANK except that $DEBLANK strips the resulting string of leading and trailing blanks, and $SUBSTR does not.

Syntax

The format of the $SUBSTR function is:

```
$SUBSTR(string, position, length)
```

where:

- *string* is the string from which the substring is derived.
- *position* is the position in the string at which the substring is to begin.
- *length* is the maximum length of the substring. If this argument is omitted, a default value of 255 is used.

The position and length arguments are rounded to positive integers.

Example 1

```
$SUBSTR('KITTREDGE', 4,3) equals 'TRE'
$SUBSTR('ANTELOPE', 7) equals 'PE'
$SUBSTR('TOO', 19, 2) equals " (null string)
$SUBSTR('ACCOUNT', -3, 5) equals 'ACCOU'
```

Example 2

This request searches the FIRST NAME field of all the Smiths and creates a list of names that begin with A, B, C, or D.

```
BEGIN
FIND.RECS: FIND ALL RECORDS FOR WHICH
```

```
                   LAST NAME = SMITH
              END FIND
              FOR EACH RECORD IN FIND.RECS
                   IF $ONEOF($SUBSTR(FIRSTNAME, 1, 1),-
                      'A/B/C/D','/')
                   THEN PLACE RECORD ON NAME
                     .
                     .
                     .
```

$SUBSYS

The $SUBSYS function determines the status of a subsystem. This function typically is used when designing applications through the Subsystem Management facility. You can use $SUBSYS to determine whether a subsystem is active before transferring control from one subsystem to another. For more information about subsystems, refer to Chapter 23.

Syntax

$SUBSYS(*subsystemname*)

$SUBSYS takes a subsystem name as an argument and returns a value indicating the status of that subsystem

Value	The subsystem is…
0	Not active (not started)
1	Active (started)
2	Draining (the STOP command has been issued and users are in the subsystem).
3	In test mode.

The $SUBSYS function without an argument returns the name of the subsystem you are currently in. For example, the User Language statement:

```
%CURSYS = $SUBSYS
```

assigns the name of the current subsystem to %CURSYS, if executed within a subsystem, or returns null if at command level.

Example

```
        .
        .
        .
 TRANSFER: IF $SUBSYS('AUTOS') = 1 THEN
 .
 *
 *              TRANSFER TO INSURANCE SUBSYSTEM
 *
                .
                .
                .
```

$TCAMFHP

The $TCAMFHP is meaningful only for users of TCAM 3270s (IODEV 21), which are no longer supported by CCA. See Appendix A for detailed information.

$TIME

Function

Returns the current time of day in the format specified.

Syntax

```
$TIME(time-format,'first-delimiter',
      'second-delimiter')
```

Where

Argument	Value	Specifies	Default value
time-format	1	Returns the time of day as *hhdmmdss* (*d*=delimiter)	1
	2	Returns the time of day as *hhdmmdssdttt* (*d*=delimiter)	
first-delimiter	1 character only	Character to place between time units HH, MM, and SS	: (colon)
second-delimiter	1 character only, second-delimiter	Character to place between time units SS and TTT	. (period)

The *second-delimiter* applies only if *time-format* is 2. You must enclose the delimiters with single quotation marks.

You can suppress either the *first-delimiter* or *second-delimiter* character by using a single quoted null string.

Examples

$TIME(*argument*)	**Returns…**
$TIME	HH:MM:SS
$TIME(1)	HH:MM:SS
$TIME(1,’-’)	HH-MM-SS
$TIME(1,’ ’)	HH MM SS
$TIME(1,”)	HHMMSS
$TIME(2)	HH:MM:SS.TTT
$TIME(2,’-’)	HH-MM-SS.TTT
$TIME(2,”,”)	HHMMSSTTT

$UNBIN

The $UNBIN function converts a value from its fixed-point binary form to the corresponding character string representation. The $UNBIN function reverses the effect of the $BINARY function.

Syntax

The format of the $UNBIN function is:

```
$UNBIN(value [,scale])
```

where:

- *value* is a string argument. This argument must be either two or four bytes long; its value is interpreted as a bit string. If the length of the argument is not two or four bytes, Model 204 issues a counting error message and returns a null string.
- *scale* indicates the number of fractional digits (bits) in the specified value. If the scale argument is omitted, a default value of 0 (integer) is used.

$UNBLANK

The $UNBLANK function returns the contents of an argument, removing leading and trailing blanks, and compressing multiple embedded blanks to one blank character. $UNBLANK is useful in conjunction with $READ. The user can respond to prompts for retrievals or comparisons in free form, without regard for leading, trailing, or embedded blanks. See the second example.

Example 1

```
$UNBLANK(’ JOHN JONES ’) equals ’JOHN JONES’
```

```
$UNBLANK('WASH., D.C.') equals 'WASH., D.C.'
```

Example 2

```
BEGIN
            %A = $UNBLANK($READ('ENTER VALUE'))
FIND.RECS:  FIND ALL RECORDS FOR WHICH
                FIELD = %A
            END FIND
              .
              .
              .
```

$UNFLOAT

The $UNFLOAT function converts a floating-point number from the standard IBM floating point format to the corresponding character string representation. Model 204 maintains 15 significant decimal digits of precision for 8-byte floating-point numbers and 6 significant digits of precision for 4-byte floating-point numbers. For an expanded discussion of rounding numbers, please refer to “Mapping and precision adjustment” on page 31-3.

Syntax

The format of the $UNFLOAT function is:

$UNFLOAT(*number*)

where *number* can be either four bytes for a single-precision floating-point number or eight bytes for a double-precision floating-point number.

$UNFLOAT converts floating-point numbers from the internal form used by Model 204 for storage efficiency to a printable string. $UNFLOAT also allows the manipulation of these numbers by User Language.

Example

For example, the result of the expression:

```
$UNFLOAT($FLOAT('1.234'))
```

is the string 1.234. If the string argument is omitted or invalid, a null string is returned.

$UNPACK

The $UNPACK function converts data in packed decimal format into a string that can be stored in a Model 204 file and processed by a request.

Syntax

The format of the $UNPACK function is:

```
$UNPACK(value [,scale])
```

where:

- *value* is the data to be unpacked. If the data is not a valid packed decimal string of as many as 18 digits, a null string is returned.
- *scale* specifies the number of implied decimal places in the packed input. If scale is greater than 18 or less than 0, a null string is returned. If scale is omitted, a default value of zero is used, indicating an integer value.

The resulting unpacked string is preceded by a minus sign if the packed number was negative. If the scale provided is greater than the number of digits in the value, the value is right-justified by the appropriate number of digits, the decimal point is inserted in the appropriate location, and a leading zero is inserted in the units position.

Leading zeros to the left of the decimal point and trailing zeros to the right of the decimal point are truncated unless the result is zero. If there are no significant digits to the right of the decimal point, the decimal point is removed from the result as well.

Examples

These examples assume that %X contains a packed value of X‘001234500C’.

Example 1:

```
%A = $UNPACK(%X,2)
```

results in %A having a string value of 12345.

Example 2:

```
%A = $UNPACK(%X,4)
```

results in %A having a string value of 123.45.

Example 3:

```
%A = $UNPACK(%X,10)
```

results in %A having a string value of 0.00012345.

$UNPOST

Function

Resets a specified Event Control Block (ECB) to an unposted state, meaning that the event has not yet occurred or has not recurred. Resets the post code to zero.

Syntax

```
$UNPOST(ECB-number)
```

Where:

ECB-number is a numeric value from one to the NECBS parameter that identifies the ECB to be unposted. The *ECB-number* can be expressed as a numeric literal, a literal enclosed in quotation marks, a %variable, or a field name.

Usage

Use the $UNPOST function to unpost a specified, numbered ECB. An ECB must be unposted before another post or wait on this ECB takes place. The $POST and $WAIT functions do not unpost an ECB, so that a subsequent wait on the same ECB cannot take place, because the ECB is still posted from the previous posting.

Use the $POST, $UNPOST, and $WAIT functions to coordinate processing between threads for numbered ECBs. For the extended quiesce ECB, QZSIG, only the $WAIT and $POST functions are valid.

$UNPOST does not reset the contents of the post code or string data set by the $POST or $ECBDSET functions for the specified ECB.

Note: Even in a non-multiprocessing environment, the $UNPOST function should be used with extreme care. Unposting an ECB while users are waiting on it may keep users in a wait state forever or until the next $POST function is issued.

The following return codes apply to the $UNPOST function:

Return code	Meaning
0	Success
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The first argument is less than one or greater than the NECBS parameter
5	NUSERS = 1

Return code	Meaning
8	No argument specified
12	Invalid argument CPQZ or invalid argument following QZSIG

$UNQREC

The $UNQREC function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit and the UNIQUE field attribute.

If $UNQREC is invoked from an ON FCC unit following the detection of a uniqueness violation, it returns the record number of the record already containing the field name = value pair. At all other times, $UNQREC returns -1.

Use $UNQREC in conjunction with $UPDREC to find the record number of the unique record already stored ($UNQREC) and the record number of the non-unique record you are attempting to store ($UPDREC). The $UNQREC function takes no arguments.

$UPCASE

The $UPCASE function translates a lowercase or mixed-case string into an uppercase-only string. The translation affects only the uppercase letters of the specified language.

Syntax

The format of the $UPCASE function is:

```
$UPCASE(string [, language-name])
```

where:

- *string* represents the string to be verified. *string* must be one of:
  - A quoted literal.
  - A %variable.
  - An unquoted field name, in which case the current value of the field is verified. In this case, the function call must be embedded in a FOR EACH RECORD loop.
- *language-name* (optional) specifies the language to use. Options are:
  - Omitting this argument, which instructs Model 204 to perform the validation for U.S. English, even if the value of the LANGUSER parameter is not NLANG.
  - A quoted asterisk (‘*’), which instructs Model 204 to use the value of the LANGUSER parameter to determine which language to use.
  - The quoted literal name of a valid language, for example, NLANGFR1 for French Canadian, Version 1. The request is cancelled with an error

message if the name is not present in NLANG$.

Example

```
$UPCASE('Name and address')
```

returns the string NAME AND ADDRESS, using U.S. English.

$UPDATE

The $UPDATE function returns the name of the group update file (in group context) or the current file (in file context). If the file is remote, $UPDATE also returns the location of the file (in the form `filename AT location`). If no group update file is defined, $UPDATE returns a null (zero-length) string.

Syntax

The format of the $UPDATE function is:

```
$UPDATE [(name)]
```

where the *name* argument overrides the default file or group context for the function.

The format for this argument is:

$UPDATE can be used to indicate the name of the update file in the current group, but the name argument must not be specified. In addition to its use in arithmetic expressions and PRINT specifications, $UPDATE also can be used as the file name in an IN clause. $CURFILE and $UPDATE are described in further detail in Chapter 16.

If the file name/group name argument is omitted or null, the default context is the context of the statement that contains the function.

Note: You cannot use an IN clause that includes both MEMBER and $UPDATE. See “IN GROUP MEMBER clause” on page 16-27 for information.

Beginning with Model 204 V4R1.0 $UPDATE compresses consecutive spaces to one space. For example, a pre-V4R1.0 $UPDATE might return:

```
'OWNERS   AT DALL'
```

V4R1.0 returns:

```
'OWNERS AT DALLAS'
```

$UPDFILE

Use the $UPDFILE function in conjunction with the ON FIELD CONSTRAINT CONFLICT unit.

If $UPDFILE is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns the name of the file in which the constraint violation occurs. If the file is remote, $UPDFILE also returns its location (in the form `filename AT location`). At all other times, $UPDFILE returns a blank. The $UPDFILE function takes no arguments.

$UPDFLD

Use the $UPDFLD function in conjunction with the ON FIELD CONSTRAINT CONFLICT unit.

If $UPDFLD is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns the name of the field in which the constraint violation occurs. At all other times, $UPDFLD returns a blank. The $UPDFLD function takes no arguments.

$UPDLOC

The $UPDLOC function is used with Parallel Query Option/204 to determine the location name of the current update unit.

$UPDLOC takes no arguments, and returns a string indicating the location of the update unit as follows:

- If there is no update unit currently in effect, the string is null.
- For a local update, the return string is ‘LOCAL’.
- For a remote update, the return string is the location name of the node where the update is occurring.

Example

```
%X = $UPDLOC
IF %X <> '' THEN
   PRINT 'UPDATE UNIT IS IN PROGRESS AT LOCATION '
WITH %X
END IF
```

$UPDOVAL

The $UPDOVAL function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit.

If $UPDOVAL is invoked from an ON FCC unit following the detection of an AT-MOST-ONE field-level constraint conflict, it returns the value of the original field occurrence which is causing the constraint violation. At all other times, $UPDOVAL returns a blank. The $UPDOVAL function takes no arguments.

$UPDREC

The $UPDREC function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit. Use $UPDREC in conjunction with $UNQREC to find the record number of the unique record already stored ($UNQREC) and the record number of the non-unique record you are attempting to store ($UPDREC).

If $UPDREC is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns the file-relative number of the record whose update causes the conflict. In all other cases, $UPDREC returns a -1. The $UPDREC function takes no arguments.

$UPDSTAT

The $UPDSTAT function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit and field attributes which have field-level constraints (UNIQUE and AT-MOST-ONE). You *must* use $UPDSTAT in the ON FCC unit when writing procedures for files which have (or might have in the future) more than one type of field-level constraint defined. The $UPDSTAT function takes no arguments.

If $UPDSTAT is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns a numeric value denoting the type of conflict that has occurred:

Value	**Meaning**
0	No violation occurred
1	A uniqueness violation occurred
2	An AT-MOST-ONE violation has occurred

$UPDSTMT

The $UPDSTMT function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit.

If $UPDSTMT is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns the type of User Language statement causing the conflict. The possible values $UPDSTMT returns are:

- ADD
- CHANGE
- INSERT
- STORE

In all other cases, $UPDSTMT returns a blank. The $UPDSTMT function takes no arguments.

$UPDVAL

The $UPDVAL function is used in conjunction with the ON FIELD CONSTRAINT CONFLICT unit. If $UPDVAL is invoked from an ON FCC unit following the detection of a field-level constraint conflict, it returns the field value which is causing the constraint violation. At all other times, $UPDVAL returns a blank. The $UPDVAL function takes no arguments.

$USER

The $USER function returns a 5-character string equal to the user's user number. In an n-terminal system, User 0 is user number 00000 and the terminal users are numbered 00001 through *n*. The value of $USER is always unique to a user during a terminal session. The value of $USER remains the same from one session to another only in installations with hard-wired BTAM terminals or static SNA Communications Server or CMS threads. This function takes no arguments.

$USERID

The $USERID function returns a variable-length character string equal to the user ID under which the user is logged into Model 204. If the login feature is not in use, $USERID returns the string NO USERID.

The USERID parameter returns the same value as the $USERID function. See the *Model 204 Command Reference Manual*. The $USERID function takes no arguments.

Example

```
BEGIN
SET HEADER 1 'MONTHLY SALES REPORT' -
     AT COLUMN 10
SET HEADER 2
SET HEADER 3 'PREPARED BY: ' AT COLUMN 10 -
     WITH $USERID
        .
        .
        .
```

$USRPRIV

The $USRPRIV function is used to test whether a user ID has been granted specific Model 204 privileges.

Syntax

The format of the $USRPRIV function is:

$USRPRIV(*privilege*,*logging option*)

where:

- *privilege* is the privilege that is to be validated. Privilege can be one of the following values:
  - ANY_ADMINISTRATOR

    The ANY_ADMINISTRATOR privilege test verifies that the user is user zero or a system manager.
  - CHANGE_FILE_PASSWORD
  - CHANGE_LOGIN_PASSWORD
  - OVERRIDE_RECORD_SECURITY
  - SUPER_USER
  - SYSTEM_ADMINISTRATOR
  - SYSTEM_MANAGER
- *logging option* specifies whether Model 204 should indicate that an error message should be issued for security violations:
  - LOG indicates that any privilege violation is logged.
  - NOLOG indicates that the privileges should be determined but any violation found is not logged.

  Currently, the logging option affects only the Security Server (formerly RACF) or Top Secret interface. The option has no effect on Model 204; the ACF2 Interface always logs a security violation regardless of the logging option.

  LOG is the default if a logging option is not specified.

$USRPRIV returns a numeric true/false value indicating the result of the authorization check as follows:

Value	User is…
0	Not authorized for the privilege or an unknown privilege name is specified.
1	Authorized for the specified privilege.

Example

The following statement could be used to test if the current user ID is authorized as a system manager.

```
IF $USRPRIV('SYSTEM_MANAGER','NOLOG') THEN
    .
    .
    .
* PERFORM SYSTEM MANAGER AUTHORIZED CODE
END IF
* ELSE UNAUTHORIZED FOR SYSTEM MANAGER FUNCTIONS
```

$VALIDATE_NUMERIC_DATA

Function

Detects invalid numeric data when data is read into an image from an external source.

Syntax

```
$VALIDATE_NUMERIC_DATA('%variable'
   | '%imagename:itemname', s1, s2, s3, s4, s5)
```

Where:

- *%variable* is a string expression enclosed in single quotation marks that resolves to the name of an item in an image. The value of this string is the standard format for referencing an image item: %*imagename:itemname,* which is also enclosed in single quotation marks.
- If the argument identifies an array item, up to five subscripts can be used, *s1-s5*. If an array is not specified, these arguments are ignored. On an array, as many of these parameters are used as the number of dimensions on the array. If any needed subscript parameter is not specified, then a value of 1 is used.

Example

If *%imagename:itemname* identifies an array, then you must identify the particular array element to validate. To do so, note the number of subscripts of the array, and specify the same number of optional subscript arguments (the s1–s5 arguments). For example, to validate %IMAGE:ARRAY(1,2), code

```
%RC=$VALIDATE_NUMERIC_DATA('%IMAGE:ARRAY',1,2)
```

Usage

The image item whose name is passed is looked up in the image definition.

- If the item is a numeric type, the contents of the data area of the image are checked to see if the data that corresponds to the item is in the correct format.
- If the item is an array, the element identified by the subscripts is tested.

The numeric return code values have the following meanings:

Return code	Means…
0	Return code zero indicates good data. Either item is numeric and the data is of the correct format, or the item is not numeric (no validation performed).

Return code	Means…
1	Item is numeric and the data is invalid.
2	On an array element, at least one subscript is out-of-bounds.
3	Item named does not exist.

$VERIFY

The $VERIFY function returns a 1 (true) if every character contained in the string is present in the second string. Otherwise, 0 (false) is returned. If the second argument is a null string, 0 is returned. If the first argument is a null string but the second argument is not, a 1 is returned.

$VERIFY can be used whenever $ALPHA, $ALPHNUM, and $VNUM do not provide sufficient flexibility.

Example 1

```
$VERIFY(’1975’,’0123456789’) equals 1
$VERIFY(’-19.5’,’0123456789+-.’) equals 1
$VERIFY(’12A’,’0123456789’) equals 0
```

Example 2

```
$VERIFY(’ABC’,") equals 0
$VERIFY(",") equals 0
$VERIFY(",’1234567890’) equals 1
```

$VIEW

The $VIEW function returns:

- The value of a parameter to the request. While $VIEW works for all viewable Model 204 parameters in local context, it does not work for some file parameters in remote context.
- File specific information
- Statistics fields. All Since Last, User, and System statistics fields can be viewed by name.

Syntax to view a parameter value

The format of the $VIEW function to view parameter values is:

$VIEW(*parameter*[,*name*])

where:

- *parameter* is a character string that represents the name of a viewable Model 204 parameter or other information. For example:

  ```
  $VIEW('ERASE')
  ```

  returns a value of @ if the ERASE parameter is set to the character @. If the parameter name is invalid or the user is not authorized to display parameter values for the file, a null string is returned.

- *name* overrides the default file or group context for the function. The format for this argument is:

  ```
  [[FILE | [PERM | TEMP] GROUP] name | $CURFILE

    | $UPDATE] [AT location]
  ```

  If this argument is omitted or null, the default context is the context of the statement containing the function. Thus, a default context is required for this function.

  You can specify $CURFILE as the *name* argument in place of a file/group name.

Viewing file-specific information

In addition to viewable parameters, the following file-specific information can also be viewed with $VIEW:

Name used with $VIEW	Description
DTSLBOPR	Date and time transaction was backed out during roll forward (for TBO files) Date and time partial transaction was applied during roll forward (for non-TBO files)
DTSLCHKP	Date and time of last checkpoint
DTSLRCVY	Date and time of last recovery
DTSLUPDT	Date and time of last update
DTSLDKWR	Date and time of last DKWR
DTSLRFWD	Date and time of last roll forward
FIWHEN	Date and time FISTAT was reset
FIWHO	Terminal ID that reset FISTAT

For example:

```
$VIEW ('FIWHEN')
```

returns:

```
FIWHEN   94.257 SEP 14 16.40.45. DATE AND TIME FISTAT WAS RESET
```

Syntax for viewing statistics fields

The first argument is the category, SLSTATS, SYSSTATS, or USRSTATS. The second argument is the name of the Statistic field to return. See the *Model 204 System Managers Guide* for the name of the statistics fields.

```
$VIEW(category, field-to-return)
```

Where the following categories and fields are valid:

Table 27-6. Valid categories and fields with $VIEW

Category	Statistics field-to-return	Equivalent-to or returns-value-of
‘SLSTATS’	‘USERID’	$USERID
‘SLSTATS’	‘ACCOUNT’	$ACCOUNT
‘SLSTATS’	‘SUBSYSTEM’	$SUBSYS with no argument
‘SLSTATS’	‘LAST’	LAST=field
‘SLSTATS’	‘PROC-FILE’	Current procedure file
‘SLSTATS’	‘PROC’	Current procedure
‘SLSTATS’	‘sl statistics name’	Statistics field
‘SYSSTATS’	‘system stat name’	Statistics field
‘SYSSTATS’	‘DKBM stat name’	Statistics field
‘USRSTATS’	‘user stat name’	Statistics field
‘USRSTATS’	‘conflict ctr name’	Statistics field

Usage

The time related Since Last statistics fields, CNCT, CPU, SCHDCPU, and STCPU, are not valid during evaluation time.

Specifying an invalid statistics field name returns a null value with the message:

```
M204.1463: INVALID PARAMETER: fieldname
```

$VNUM

The $VNUM function returns a 1 if the given argument is in a valid format for a SORT BY VALUE NUMERICAL statement or for any type of mathematical operation. To be valid, the argument must be a quoted rational number with optional sign. Leading or trailing blanks, or blanks between the sign and the number, are ignored. If the contents of the argument do not have the required form, 0 is returned. Numeric values with more than 63 significant digits are not in the correct form; a 0 is returned.

Syntax

```
$VNUM(numeric-string-value{,'SORT' | 'SORTKEY'

     | 'FLOAT' | 'BINARY'})
```

Where:

- *SORTKEY* returns 1 if the numeric value is valid as a numeric string sortkey. See “Proper form required” on page 9-3. A numeric string sortkey value that is not acceptable to $VNUM as a *SORTKEY* value is sorted in character order even if the SORT statement specifies numeric order.

  The following compression rules apply to SORTKEY:

  – Leading plus (+) or minus (-) sign is compressed.

  – Leading blanks before and after the (optional) leading +/- sign are compressed.

  – Leading zeros after the (optional) leading +/- sign and blanks are compressed.

  – 1–63 integer digits before the (optional) decimal point are allowed.

  – Values greater than -1 and less than 1 require leading “0.*nnn*”.

  – Optional decimal point and fractional value are allowed.

  – Trailing blanks are compressed.

  – 1–253-byte total length is allowed.

- *’FLOAT’* returns 1 if the numeric value is valid as an E-format floating point numeric string.

- *’BINARY’* returns 1 if the numeric value is valid as a compressible binary value (to be stored in a BINARY NON-CODED field). See “Storing values in BINARY fields” on page 15-17. Otherwise returns a 0.

  The following compression rules apply to BINARY:

  – 1–9 decimal integer values are allowed.

  – No leading zeros are compressed.

  – Leading plus (+) sign is compressed.

  – Leading minus (-) sign is allowed.

Example 1

```
$VNUM(' + 256.73 ') equals 1
$VNUM('14') equals 1
$VNUM('-17.17') equals 1
$VNUM('.1794763') equals 1
```

Example 2

```
$VNUM(' -256.73 AB') equals 0
```

```
$VNUM('256.73-') equals 0
$VNUM(' TWELVE') equals 0
```

Example 3

This example averages premium amounts. The amounts are included in the average only if they meet the $VNUM specifications.

```
BEGIN
          %CT IS FLOAT
GET.RECS: FIND ALL RECORDS
          END FIND
          FOR EACH RECORD IN GET.RECS
              IF $VNUM(TOTAL PREMIUM) THEN
                  %TOT = %TOT + TOTAL PREMIUM
                  %CT = %CT + 1
              END IF
          END FOR
          %AVERAGE = %TOT/%CT
          PRINT 'THE AVERAGE PREMIUM IS ' -
              WITH '$' WITH %AVERAGE
END
```

$WAIT

Function

Suspend a user until an Event Control Block (ECB) is posted

Syntax

$WAIT(*ECB-number* [,'<u>SWAP'</u> | 'NOSWAP']

    [,time-interval] | 'CPQZ' | 'QZSIG')

Where:

- *ECB-number* is a string with a numeric value from one to the NECBS parameter, which identifies the ECB upon which to wait. *ECB-number* can be expressed as a numeric, a literal, a %variable, or a field name. This is a user defined event. When these users are placed in a wait state, the wait type is 30.
- SWAP keyword specifies a swappable wait; this is the default. The NOSWAP keyword specifies a nonswappable wait. If you enter a keyword, enclose it in single quotation marks.
- *time-interval* specifies the maximum number of seconds to wait. A wait that is finished, because the time expired, is indicated by a return code 16.

A *time-interval* may be from zero to 86400 and indicated as a number, %variable, or a string. A *time-interval* less than one is treated as no time-interval supplied with no time-out to happen.

If a *time-interval* is specified, the second argument, SWAP or NOSWAP, is activated. You can specify ’SWAP’ or ’NOSWAP’, or accept the default, SWAP, using the following syntax:

```
$WAIT(ECB-number,,time-interval)
```

The *time-interval* can be used as an argument to $WAIT to determine when an extended quiesce event starts; at that time the backup can be submitted. When these users are placed in a wait state, the wait type is 47.

- The CPQZ and QZSIG keywords are the named ECBs for extended quiesce, which are used by the NonStop/204 facility for independently run third-party backups. See the *Model 204 System Manager’s Guide* for an explanation of the facility.
  - CPQZ is posted internally at the beginning of each extended quiesce and unposted internally at the interval end.
  - QZSIG can be posted during the extended quiesce. It is unposted internally at the end of extended quiesce.

  The $WAIT function applied to CPQZ and QZSIG are bumpable, swappable waits of type 47 for CPQZ and 48 for QZSIG. You cannot specify the SWAP or NOSWAP keywords or *time-interval* with the named ECBs.

Usage

You can use the $WAIT function to suspend a user, meaning: put that user into a wait state until the ECB is posted by another user with the $POST function. Users who have issued a $WAIT function call are bumpable and may be swappable depending on whether SWAP or NOSWAP was used in the $WAIT call.

Caution: Limit the use of the $WAIT function with the NOSWAP option to situations where only a small number of threads may use it. This will avoid having all servers occupied by users in a NOSWAP state and having no available server for a posting user to swap into.

Using the $WAIT function, you can put a User Language thread into a wait state. To perform third-party backups, the thread must wait for the extended quiesce of a checkpoint to start and then submit a backup job.

- For numbered ECBs, you can use the $POST, $UNPOST, and $WAIT functions to coordinate processing between threads.
- For the QZSIG ECB, you can use the $POST and $WAIT functions to signal and recognize the end of an extended quiesce for third-party backups.

- For the CPQZ ECB, you can use the $WAIT function to wait on the start of an extended quiesce.

The following return codes apply to the $WAIT function:

Return code	Meaning
0	Success
2	Bad argument specified
3	NECBS parameter is not specified or is zero
4	The first argument is less than one or greater than the NECBS parameter
5	NUSERS = 1
6	NOSWAP and NSERVS EQ 1
8	No argument specified
9	Checkpointing inactive, if using extended quiesce ECBs, CPQZ or QZSIG
11	The CPQZ or QZSIG ECB is already posted
12	Invalid argument CPQZ or invalid argument following QZSIG
13	For QZSIG, the system is not in extended quiesce or already leaving extended quiesce
15	Time interval is not numeric or greater than 86,400
16	$WAIT finished due to expired time interval
17	Second argument entered is not SWAP or NOSWAP

When the ECB specified in the $WAIT call is posted, the waiting user will resume evaluation and may capture the post code with $STATUSD. See the following $WAIT example.

Example

The following code illustrates an interaction between User 1 and User 2. User 1 issues the following:

```
BEGIN
  %X=$ECBDSET(3,'THIS IS ECB 3')
  PAUSE 60
  %X=$POST(3,5678)
END
```

User 2 starting after the previous $ECBDSET, but before 60 seconds have elapsed:

```
BEGIN
PRINT 'WAITING ON ECB 3'
```

```
NP
%X=$WAIT(3)
PRINT ’ECB 3 HAS BEEN POSTED WITH POST CODE= ’ WITH -
      $STATUSD
END
```

User 2 will be suspended (WT=30) when the $WAIT call is evaluated and will resume processing when ECB number 3 is posted by User 1. $STATUSD will return the post code value = 5678.

If User 2 starts after 60 seconds have elapsed and the ECB number 3 has been posted, then User 2 will not wait, but will print the last post code for the ECB number 3.

$WORD

The $WORD function searches a string from the left for a complete word and returns the *n*th word in a specified string, delimited by a blank or optionally specified character.

Syntax

The format of the $WORD function is:

```
$WORD('inputstring' , [϶δelimiter'] , [n])
```

where:

- *inputstring* is the input from which the specified word is to be extracted. *inputstring* can be a quoted literal or a %variable.
- *delimiter* is an optional quoted character or string to be used as a delimiter in parsing the input string into words. If *delimiter* is not specified, it defaults to a blank space. If *delimiter* consists of more than one character, Model 204 uses the first character in the string as the delimiter.
- *n* is the ordinal number of the word to be extracted. $WORD returns a null string under any of the following conditions:
  - *n* is not specified
  - *n* is less than 1 (0 or negative)
  - *n* is greater than the number of words in *inputstring*

The commas which separate the three arguments are required, even if *delimiter* is not specified.

Usage

Null values between delimiters are not considered words. For example,

```
$WORD(’=ABCD==D=XXX’,’=’,2)
```

returns D as the second word, not a null value.

Examples

The following function returns the third word in the string, "THE", when a single space delimits a word:

```
$WORD (’NOW IS THE TIME’,,3)
```

The following function returns the second word in the string, “S THE T", where a word is composed of the characters that precede a delimiting I:

```
$WORD (’NOW IS THE TIME’,’I’,2)
```

$WORDS

The $WORDS function returns the number of words in a specified string, delimited by a blank or optionally specified character.

Syntax

The format of the $WORDS function is:

`$WORDS(’`*`inputstring`*`’, [϶δ`*`elimiter’`*`])`

where:

- *inputstring* is the input from which the specified word is to be extracted. *inputstring* can be a quoted literal or a %variable.
- *delimiter* is an optional character to be used as a delimiter in parsing the input string into words. If *delimiter* is not specified, it defaults to a blank space.

Usage

Null values between delimiters are not considered words. For example,

```
$WORDS(’ABC==D=XXX’,’=’)
```

returns a count of three words, not four.

Example

The following function returns 4, the number of words in the string when a blank delimits a word:

```
$WORDS (‘NOW IS THE TIME’)
```

$X2C

The $X2C function changes a 2-byte character of input into 1-byte hexadecimal-equivalent EBCDIC characters. Called with a character string, $X2C returns a character string that is half as long. The maximum input length is 255 bytes. If the input length is more than 255 bytes or if input contains invalid hexadecimal data, a null string is returned. There is no function to translate 2-byte hexadecimal characters to ASCII characters.

The output is one half the length of the input. Each pair of hexadecimal characters in the output becomes an EBCDIC character. For example, C‘01’ = X‘01’ or E8C5E2 = ‘YES.’

Syntax

The format for the $X2C function is:

```
$X2C(inputchar)
```

where *inputchar* is the input character string (either a %variable or a quoted literal) to be converted to one-byte hexadecimal-equivalent characters.

Input must be an even number of bytes (divisible by 2) and contain only combinations of the following characters:

```
0123456789ABCDEF
```

If the input character string is invalid for any reason and cannot be converted, then the output string returned by the function is set to a null.

Example 1

```
BEGIN
  %INPUT_IN_HEX = ’05’
  %PAD_CHAR = $X2C(%INPUT_IN_HEX)
END
```

Example 2

```
BEGIN
PRINT $X2C(’C1C2C3’)
END

(output is ABC)
```

Example 3

```
BEGIN
PRINT $X2C(’F1’)
END

(output is 1)
```

Mathematical functions

The mathematical functions are an optional feature of Model 204 that might not be available at every installation. If the option was not installed, attempts to use the mathematical functions generate compile-time errors. The notation |x| indicates the value of the argument x, rounded to the nearest integer.

Using Language Environment mathematics $functions

Beginning with Version 5.1, if you want to use the Model 204 mathematics $functions, you can install either the FORTRAN runtime libraries or use the IBM LE runtime libraries. Model 204 now includes an interface to the LE libraries. As in prior releases of Model 204, you are not required to use the mathematics $functions. Consult the Model 204 installation guide for your operating system.

CCA recommends that you use the FORTRAN library for applications depending on mathematical functions performance.

Precision

The Model 204 internal numeric data representation can maintain 15 significant digits of accuracy. However, some of IBM’s FORTRAN routines use algorithms that are not accurate to 15 places. You should be aware that some of the low-order digits returned by these routines might not be meaningful.

Error handling

If invalid numerical values are passed to the mathematical functions (for example, a negative number to $SQRT), an error message is printed at the user’s terminal and 0 is returned as the function’s value.

$ABS(x)

The $ABS function returns the absolute value of x. For example:

```
$ABS(-50) = 50
$ABS(6) = 6
```

$ARCCOS(x)

The $ARCCOS function returns the value of the arc cosine of x in radians. If the magnitude of x exceeds 1, an error message is printed and a 0 is returned.

$ARCSIN(x)

The $ARCSIN function returns the value of the arc sine of x in radians. If the magnitude of x exceeds 1, an error message is printed and a 0 result is returned.

$ARCTAN(x)

The $ARCTAN function returns the value of the arc tangent of x in radians.

$ARCTAN2(x,y)

The $ARCTAN2 function returns the value arctan(x/y) in radians. If the second argument is 0 (or omitted), an error message is printed and a 0 is returned.

$COS(x)

The $COS function returns the value of the cosine of x in radians. If the argument exceeds 10^{15} radians, an error message is printed and a 0 is returned.

$COSH(x)

The $COSH function returns the value cosh(x) in radians. If the argument exceeds the value 175.366, an error message is printed and a 0 is returned.

$COTAN(x)

The $COTAN function returns the value cotan(x). If the magnitude of the argument exceeds 10^{15} radians, an error message is printed and a 0 is returned.

$ERF(x)

The $ERF function returns the value:

$$\frac{2}{\sqrt{\pi}}\int_{0}^{\alpha} e^{-z^2}\,dz$$

$ERFC(x)

The $ERFC function returns the value:

$$1-\frac{2}{\sqrt{\pi}}\int_{0}^{\alpha} e^{-z^2}\,dz$$

$EXP(x)

The $EXP function returns the value e. If x exceeds 174.63, an error message is printed and a 0 is returned.

$GAMMA(x)

The $GAMMA function returns the value:

$$\int_0^\infty u^{x-1} \quad e^{-u} \, du$$

If x is not within the range 0 < x < 57.5744, an error message is printed and a 0 is returned.

$IXPI(x,y)

In the $IXPI function, both arguments are rounded to the nearest integral values, and the value |x| is raised to the |y| power. If |x| equals 0 and |y| is less than or equal to 0, an error message is printed and a 0 is returned. For example:

$IXPI(8,2) = 8^2 = 64
$IXPI(2.4,.5) = $IXPI(2,1) = 2

$LGAMMA (x)

The $LGAMMA function returns the value:

$$\log_e \int_0^\infty u^{-x-1} \quad e^{-u} \, du$$

If x is not in the range 0 < x < 4.2913 * 10^{73}, an error message is printed and a 0 is returned.

$LOG(x)

The $LOG function returns the natural logarithm (the logarithm base e) of a number x. If x is not greater than 0, an error message is printed and a 0 is returned.

$LOG10(x)

The $LOG10 function returns the logarithm base 10 of a number x. If x is not greater than 0, an error message is printed and a 0 is returned.

$MAX(X1, X2, X3, X4, X5)

The $MAX function returns the highest value in a list of as many as five arguments. For example:

```
$MAX(-6, 5, 0, 4, 3) = 5
$MAX(-6,4) = 4
$MAX(4,-6,70.3) = 70.3
$MAX(-6, ,-5) = -5
$MAX(-6,0,-5) = 0
```

Omitted arguments to $MAX are ignored.

$MIN(X1, X2, X3, X4, X5)

The $MIN function returns the value of the smallest argument in a list of as many as five arguments. For example:

```
$MIN(-6, 5, 0, 4, 3) = -6
$MIN(-4, -7, 2) = -7
$MIN(4, ,2) = 2
$MIN(4, 0, 2) = 0
```

Omitted arguments to $MIN are ignored.

Arguments to $MAX and $MIN typically are the results of other computations. For example:

```
%A = $MAX($ABS(B),$ABS(C))
```

$PI

The $PI function returns the value of π to 15 significant digits (3.14159265358979).

$RXPI(x,y)

In the $RXPI function, the second argument (y) is rounded to the nearest integer and the value of x raised to the |y| power is returned. Omitted arguments are set to 0. If x = 0 and |y| is less than or equal to 0, an error message is printed and a 0 is returned. For example:

$RXPI(2, 3) = 2^3 = 8
```
$RXPI(.5, 1.4) = $RXPI(.5, 1) = .5
$RXPI(.5, 2) = .25
```

$RXPR(x,y)

The $RXPR function returns the value x to the y power. If x < 0, or x = 0 and y is less than or equal to 0, an error message is printed and a 0 is returned. For example:

```
$RXPR(10, 2) = 10^2 = 100
$RXPR(64, .5) = 8
$RXPR(256, .25) = 4
```

$SIN(x)

The $SIN function returns the value of the sine of x. If the magnitude of x exceeds the value of 10^{15} radians, an error message is printed and a 0 is returned.

$SINH(x)

The $SINH function returns the value sinh(x). If the magnitude of x exceeds 175.366, an error message is printed and a 0 is returned.

$SQRT(x)

The $SQRT function returns the value of the square root of x. If x is negative, an error message is printed and a 0 is returned. For example:

```
$SQRT(64) = 8
$SQRT(2) = 1.41421356
```

$TAN(x)

The $TAN function returns the value of the tangent of x. If x exceeds 10^{15} radians, an error message is printed and a 0 is returned.

$TANH(x)

The $TANH function returns the value tanh(x).

s

28

Abbreviations

In this chapter

- User Language abbreviations
- Command abbreviations

User Language abbreviations

Certain User Language statements, phrases, and keywords can be abbreviated as shown in Table 28-1. These abbreviations can be used interchangeably with their unabbreviated counterparts.

Table 28-1. User Language abbreviations

Abbreviation	Statement, phrase, or keyword
ALPHA	ALPHABETICALLY
CH	CHANGE
CMMTRL	COMMIT RELEASE
CT *label*	COUNT RECORDS IN *label*
CT ON *listname*	COUNT RECORDS ON LIST *listname*
CTO *fieldname*	COUNT OCCURRENCES OF *fieldname*
FD	FIND ALL RECORDS
FD	FIND ALL RECORDS FOR WHICH
FD IN *label*	FIND ALL RECORDS IN *label* FOR WHICH
FD ON *listname*	FIND ALL RECORDS ON LIST *listname* FOR WHICH
FDR	FIND AND RESERVE ALL RECORDS
FDR	FIND AND RESERVE ALL RECORDS FOR WHICH
FDR IN *label*	FIND AND RESERVE ALL RECORDS IN *label* FOR WHICH
FDR ON *listname*	FIND AND RESERVE ALL RECORDS ON LIST *listname* FOR WHICH
FDV *fieldname*	FIND ALL VALUES OF *fieldname*
FDWOL RECORDS	FIND WITHOUT LOCKS RECORDS
FEO *fieldname*	FOR EACH OCCURRENCE OF *fieldname*
FPC	FIND AND PRINT COUNT
FR	FOR EACH RECORD
FR *label*	FOR EACH RECORD IN *label*
FR IN *label*	FOR EACH RECORD IN *label*
FR ON *listname*	FOR EACH RECORD ON LIST *listname*
FRN	FOR RECORD NUMBER
FRV *fieldname*	FOR EACH VALUE OF *fieldname*
FRV IN *label*	FOR EACH VALUE IN *label*

Table 28-1. User Language abbreviations (continued)

Abbreviation	Statement, phrase, or keyword
FS	FIELD SAVE
I procname	INCLUDE procname
INOUT	INPUT OUTPUT
NP	NEW PAGE
NUM	NUMERICALLY
OCC IN	OCCURRENCE IN
ON ATTN	ON ATTENTION
ON FCC	ON FIELD CONSTRAINT CONFLICT
ORD	IN ORDER
PAI	PRINT ALL INFORMATION
ST	STORE RECORD

Command abbreviations

In addition, certain Model 204 system control commands can be abbreviated. Table 28-2 lists the abbreviations for the commands introduced in this manual. These abbreviations can be used interchangeably with their unabbreviated counterparts.

Table 28-2. Model 204 command abbreviations

Abbreviation	Command
B	BEGIN
D	DISPLAY
I	INCLUDE
O	OPEN
PROC	PROCEDURE
U	USE
V	VIEW

Example

This first request uses abbreviations.:

```
  BGIN
  GET.RECS: IN CLIENTS FD
            END FIND
            FR GET.RECS
               PAI
            END FOR
  END
```

The second request is the same request written in the long form.

```
BEGIN
GET.RECS: IN CLIENTS FIND ALL RECORDS
          END FIND
          FOR EACH RECORD IN GET.RECS
              PRINT ALL INFORMATION
          END FOR
END
```

29

Reserved Words and Characters

In this chapter

- Rules for reserved words and characters

Rules for reserved words and characters

A number of words and characters have special meaning to Model 204 and either cannot be used as part of field names or values or can only be used as part of a quoted string.

The rules for Model 204 reserved words and characters are as follows:

- Any word or character, including the space character, can be used as part of a field name, except the following:

  ??

  ?$

  ?&

  @ (as delete character)

  # (as flush character)

  ;

  Note: The delete (@) and flush (#) characters used at an installation are controlled by the ERASE and FLUSH parameters (described in the *Model 204 Command Reference Manual*). If different symbols are chosen, the restriction on using these characters in field names applies to the new symbols.

- The following list of reserved words or operators can be part of a string-without-quotes as long as they are not surrounded by spaces (ANDIRON is acceptable while AND IRON is not). They can be part of a quoted string as long as they do not stand alone (‘A OR B’ is acceptable while ‘OR’ is not). Although field names and values can contain reserved words, requests which reference those fields might not compile, or might produce unexpected results.

Table 29-1. Reserved words and operators

AFTER	EACH	NOR	RECORDS	WITH
ALL	EDIT	NOT	TAB	EQ
AND	END	OCC	THEN	GE
AT	FROM	OCCURRENCE	TO	GT
BEFORE	IN	ON	VALUE	LE
BY	IS	OR	VALUES	LT
COUNT	LIKE	RECORD	WHERE	NE

- If any of the following reserved characters is embedded in a field name, the character must be part of a quoted string. When creating field names, CCA recommends avoiding the following characters:

Table 29-2. Reserved characters

$	>	+
(	<	- (minus sign/hyphen)
)	*	ÿ
=	/	,
...	:	%

How to refer to a field name containing reserved words or characters

As in Table 29-1 and Table 29-2, if a field name does contain a reserved word or character, you must enclose it in single quotes when referencing it in a request. For example, the following FIND statement references a field named NOR SLS:

```
FIND ALL RECORDS FOR WHICH 'NOR SLS' > 0
```

The following DISPLAY command references a procedure called %SAVINGS:

```
DISPLAY PROCEDURE '%SAVINGS'
```

30

Request Composition Rules

In this chapter

- Statement labels
- Statement block ends
- Statement format
- Field names and values
- Quotation marks

Statement labels

This chapter summarizes the rules for composing compilable User Language requests. The rules for specifying statement labels in User Language statements are summarized below.

- A statement label must begin with a letter (A–Z, a–z) which can be followed by one or more occurrences of a letter (A–Z, a–z), digit (0–9), period (.), or underscore (_). The label can be a maximum of 254 characters in length.
- A statement label should not be a User Language keyword. Model 204 interprets a statement label which is a keyword as the keyword itself, not as a statement label, if that keyword makes syntactic sense where the statement label is referenced.
- The label must be the first word on a line, must end with a colon, and must be followed by a space.
- A label can start in any column up to but not including column INCCC.
- Within a request, statement labels should be unique. If not, Model 204 displays the message:

  ```
  M204.0223: STATEMENT LABEL MULTIPLY DEFINED
  ```

  This is a warning message only; the request can still be run.

Statements that must be labeled

Any statement can be labeled, but some statements must be labeled.

In particular, a statement must be labeled if:

- It immediately follows a STORE RECORD or a FIND statement and an END STORE or END FIND statement is not being used to end the statement.   In this case, the statement label indicates the end of the preceding statement.
- It is referred to by later statements. In most requests, COUNT, FIND, FOR EACH OCCURRENCE, FOR EACH VALUE, NOTE, and SUBROUTINE, are referred to later and should therefore be labeled.

Unlabeled statements

The retrieval conditions of a FIND statement, the conditions of an IF statement and ELSEIF clause, and the fields of a STORE RECORD statement must *not* be labeled, even if they begin new lines.

If a statement is not labeled, it is assigned a default level of nesting. The default level is the same as the level of the previous statement. If there is no previous statement, the default is the first level. If the previous statement starts a loop or a THEN clause, or is a SUBROUTINE statement, the default level is one greater than the level of the previous statement.

Label references

Labels allow statements to refer to other User Language statements. The label reference must be coded exactly as the label, including upper- and lowercase lettering. However, the label reference must omit the colon.

Statement block ends

Beginning a block

The FIND (except for FIND ALL VALUES), FOR, IF, ON, REPEATE, STORE, and SUBROUTINE statements begin blocks in User Language and therefore must be explicitly ended.

Ending a block statement

You can end a block statement in the following ways:

- The appropriate block end statement (END {*statement type*} [*label*]) can be used to end any block.
- A label can be used to end blocks of multiline conditions (blocks begun by the FIND and STORE statements). A label cannot be used to end blocks of nested statements (blocks begun by the FOR, IF, ON, REPEAT, or SUBROUTINE statements).
- The END BLOCK statement can be used to end any block except for blocks of multiline conditions (blocks begun by the FIND and STORE statements).
- END, END MORE, END NORUN, and END USE can be used to end any block, because these forms the END statement terminate the request, thereby ending all statements within the request.

Statement format

The rules for constructing User Language statements are summarized below:

Begin statements on a new line

Each statement must begin on a new line unless it follows a THEN, ELSE, or ELSEIF clause.

Statement continuation

Use of hyphens

You can continue statements onto another line by using a hyphen after the last character on the line to be continued, or by using any nonblank character in the column specified by the INCCC parameter (discussed in the *Model 204 Terminal User's Guide*).

Use of parentheses

You can continue statements using parentheses, although CCA recommends that you use parentheses only to change the order of precedence in retrieval statements or with FR WHERE statements. See:

- "Interpretation of Boolean operators in retrievals" on page 4-14 for a discussion of line continuation in FIND statements
- "Specify retrieval criteria on one logical line" on page 5-7 in FR WHERE statements.

Avoid blank lines between continued lines

You should not use blank lines between continued lines. The following example illustrates two problems that occur if you use blank lines between continued lines (for example, if you wanted to double space your code).

```
B
%A = 'AAA-

BBB'
PRINT %A
IF $SETG('A','B') OR -

   $SETG('B','C') THEN
   PRINT 'ERROR'
END
```

First, when you print %A, you will get "AAA;BBB"; Model 204 inserts a semicolon to represent the blank line. The second problem is that the second $SETG statement will be rejected with an "INVALID STATEMENT" message.

To correct both problems, simply add a hyphen to each blank line following the continued line.

The following example illustrates proper coding:

```
B
%A = 'AAA-
         -
BBB'
PRINT %A
IF $SETG('A','B') OR -
                     -
   $SETG('B','C') THEN
   PRINT 'ERROR'
END
```

Compatibility issues

In Model 204 releases prior to Version 2.1, the compiler accepted certain types of invalid expressions, but returned unpredictable results. The following is an example of an invalid expression:

```
IF %X = %Y *
   %Z THEN
```

Note that the first line (IF %X . . .) needs a continuation hyphen.

Invalid expressions are recognized as syntax errors and produce the following message:

```
M204.0298: INVALID OPTION: cccc
```

where *cccc* is the program text that was in error.

Also, the following example code lacks continuation hyphen following an “AND” or an “OR” within a conditional IF statement.

```
IF FIELD1 = A AND
   FIELD2 = B AND
   FIELD3 = C THEN
     DO X
```

The above example results in a compiler error. The proper syntax for the above statement is:

```
IF FIELD1 = A AND -
   FIELD2 = B AND -
   FIELD3 = C THEN
     DO X
```

Line length

An input line, together with its continuation lines, constitute one logical line. Lines continued with a nonblank character in column INCCC cannot exceed

LIBUFF characters. Lines continued with a hyphen before the end of a terminal line have no length limit.

Blanks between words

At least one blank must separate words in a statement. Extra blanks are optional.

Where lines can begin

Typing can begin anywhere on the terminal line.

Logical lines

Most statements are entered as one logical line.

The following statements might require the use of multiple logical lines:

- FIND

  The first retrieval condition for a FIND statement can appear on the same line as the FIND clause. Other retrieval conditions which start a new line are preceded implicitly by a logical AND.

- IF and ELSEIF

  The conditional expression in an IF statement or ELSEIF clause can begin on the same line as IF or ELSEIF on the next line. Each new line in the expression is preceded implicitly by a logical AND. THEN can appear on the same line as the expression or on the next line.

- STORE RECORD

  Each fieldname = value pair of a record to be stored must be entered on a new line.

The number of input lines used by FIND, IF, ELSEIF or STORE RECORD is unlimited.

Use of semicolon to perform a carriage return

Description

The semicolon (;) normally performs a carriage return within a request. For example:

```
BEGIN;PRINT.INFO: FIND AND PRINT COUNT;END
```

Do not use a semicolon after the INCLUDE statement

A semicolon should not be used after the INCLUDE statement because INCLUDE takes the next physical line from the specified procedure. For example, the procedure GREET consists of a single line:

```
PRINT ’HELLO’
```

Therefore, the following request:

```
BEGIN
INCLUDE GREET
END
```

causes Model 204 to read the PRINT statement from procedure GREET immediately after the INCLUDE. If the same request is entered with semicolons:

```
BEGIN;INCLUDE GREET;END
```

Model 204 reads the INCLUDE statement and does what is necessary to read the next physical line from the GREET procedure. Model 204 then looks for the next logical line, which comes from the current physical line, and sees the END statement. This terminates a request which does nothing. The command handler reads the next physical line from the GREET procedure, getting PRINT ’HELLO’. PRINT is flagged as an invalid command.

The LINEND parameter

The LINEND parameter sets the logical line-end character to something other than a semicolon. If the LINEND parameter is set to a non-printable character, you cannot stack multiple logical lines on a single physical line. LINEND should never be set to X‘00’. The *Model 204 Editing Guide* discusses the effects of the LINEND parameter on the editing and including of a procedure.

Field names and values

Rules for field names

The rules for specifying field names in the User Language statements are summarized below.

1. Any word or character, including space, can be used as part of a field name except the correction characters @ and #. The choice of symbols to signify correction characters is parameter controlled and can be changed. If you chooses different symbols, the restriction regarding the use of @ and # in field names then applies to the new symbols instead. (See the *Model 204 Command Reference Manual*.)
2. If any reserved word or character is embedded in a field name, the word or character must be part of a quoted string (see the discussion on field attributes in Chapter 32).
3. Field names that contain a colon followed by a space (for example, COLOR: CAR) cannot be distinguished from labels when used as the first word on a line. Any field name containing this combination either should be renamed or must be enclosed in single quotes at the beginning of a line.
4. When more than one consecutive space appears in a field name, the extra spaces are ignored.
5. A field name can be subscripted by including a parenthesized expression after the name. This facilitates references to multiply occurring fields (see Chapter 19).
6. To refer to a fieldname indirectly, specify it with the format %%name (see the discussion in “Field name variables” on page 10-31). Field name variables also can be subscripted.

Examples

Some examples of *legal* field names are:

```
AGENT
ANNUAL_INTEREST
'TOTAL USE'
```

Some examples of *illegal* field names are:

Unacceptable	How to correct…
YEAR TO DATE	Year-to-date
USE COUNT	’Use count’
STRING@	Use another character, not @, or change the correction character
AND	"AND"

Rules for field values

The rules governing the formation of field values are the same as rules 1, 2, and 4 for field names (see the preceding discussion).

In addition, values cannot contain more than 255 characters.

Use of quotes with field values

Although single quotes are required to enclose a text field value when the value contains reserved words or characters or when used in expressions, it is good practice to enclose the text in quotes, even if it contains no reserved words or characters. To store the field:

```
PARENTS = MARY AND JOHN SMITH
```

the value portion of the pair must be quoted because AND is a reserved word. Used with a STORE RECORD statement, the field should look like this:

```
PARENTS = ’MARY AND JOHN SMITH’
```

For more information about the use of quotation marks, refer to “Quotation marks” on page 30-11.

Quotation marks

You can direct Model 204 to display arbitrary text information by means of the single quote symbol.

Uses for quotation marks

User Language statement employ quotation marks as follows:

- Following a PRINT statement, any characters enclosed in single quotes are printed literally when the statement is executed.
- Within a request, quotation marks can be used to provide titles for output or display messages to the end user.
- Quoted material can be included in a list with other things to be printed (see PRINT statement).
- Quoted material can be used in expressions.
- Quoted material can be saved by the NOTE statement for later use in a request.
- Single quotes also are used when a reserved word or symbol is to be interpreted in other than its standard system sense. Thus when reserved words or symbols are used in the formation of field names or values, the entire string must be enclosed in quotation marks.

Rules for using quotation marks

Model 204 handles quotation marks as follows:

1. A pair of single quotation marks, for example ‘TEXT’, delineates a quoted string.
2. Quoted strings are stored and utilized with quote marks dropped. Thus `PARENTS = 'MARY AND JOHN SMITH'` is stored as `PARENTS = MARY AND JOHN SMITH`, and the statement `PRINT PARENTS` results in the output `MARY AND JOHN SMITH`.
3. A pair of consecutive quotation marks inside of a quoted string is replaced by a single quotation mark when the string is stored or printed. For example, `PRINT 'FATHER''S NAME'` results in the output `FATHER'S NAME` and `PRINT ''AND''` yields `'AND'`.
4. A pair of consecutive quotation marks that is not included in a quoted string converts to a character string of zero length, called a null string.
5. Only pairs of quotation marks are used.

Example The following examples illustrate different ways of referring to a TITLE field with the value, WHY JOHNNY CAN’T READ:

```
TITLE = ’WHY JOHNNY CAN"T READ’
TITLE = WHY JOHNNY ’CAN"T’ READ
TITLE = WHY JOHNNY CANT""T READ
```

Quotation marks designating a null string

In a FIND statement, the pair:

```
BIRTHPLACE = ’’
```

is the same as:

```
BIRTHPLACE =
```

The retrieved records will contain the field BIRTHPLACE and its value will be null. If the field is printed, it will result in a blank line of output.

Quoting a reserved word

To store the field, A OR B = VALUE, one of the following must be used because OR and VALUE are reserved words:

```
A ’OR B’ = ’VALUE’
A ’OR’ B = ’VALUE’
```

A field name beginning with a quotation mark looks like quoted text to a PRINT statement. Thus the forms ‘A OR B’ and ‘A OR’ B should not be used.

After Model 204 encounters a closing quote, it does not recognize a reserved word again until a delimiter such as a space or an operator is encountered. In the following string:

```
’TESTDATA’AND
```

Model 204 regards the AND not as a reserved word but as part of the string (TESTDATAAND). The quotes do not actually have to surround the string.

31
Floating Point Conversion, Rounding, and Precision Rules

In this chapter

- Conversion
- Mapping and precision adjustment.

Conversion

Values can be stored in floating point fields using any of the file update statements (for example, STORE RECORD) or can be stored in a %variable. When the value to be stored is supplied as a string, Model 204 attempts to convert the string to floating point.

Significant digits past the fifteenth digit are ignored and are treated as zeros regardless of the precision of the value's receptacle. This applies to numbers assigned to fields or %variables or used in arithmetic expressions.

Mapping and precision adjustment

For 8-byte and 16-byte floating point numbers, Model 204 maintains 15 significant decimal digits of precision. For 4-byte floating point numbers, Model 204 maintains six significant digits of precision.

Floating point format is an approximate numeric representation that cannot always directly represent decimal values. To provide exact equality in comparisons, Model 204 maps floating point numbers to the floating point number closest in value to a 6- or 15-digit decimal number. This mapping process is, in effect, a decimal rounding process. This mapping occurs in the following cases:

1. Decimal rounding occurs after addition and subtraction arithmetic operations involving floating point numbers.
2. If the field is defined as KEY, the value to be indexed is rounded.
3. If the field is defined as ORDERED NUMERIC, the value to be indexed is rounded.
4. If the field is used in a direct Table B search in a FIND statement, both the value to be searched for and each field accessed in Table B are rounded.
5. If the field is to be used as a key in a SORT statement, the value is rounded before being concatenated to the key.

Assigning floating point numbers to floating point numbers of different lengths

Table 31-1 indicates how precision is adjusted with floating point numbers of different lengths.

Table 31-1. Floating point precision adjustment

Assignment or comparison or conversion	Precision adjustment
FLOAT 8 to FLOAT 4	Hexadecimal rounding (that is, if second half of an 8-byte floating point number has its high order bit on, a bit is added to the first half of the 8-byte number).
FLOAT 16 to FLOAT 8	Truncated.
FLOAT 4 to FLOAT 8	Decimal rounding (that is, value is converted to the 8-byte floating point number closest in value to the 6-digit decimal representation of the FLOAT 4 value).
FLOAT 8 to FLOAT 16	Padded with zeros.

32

Field Attributes

In this chapter

- File model feature
- Field attribute descriptions

File model feature

The file manager defines fields and assigns attributes to each field. This procedure is discussed in detail in the *Model 204 File Manager's Guide*. The field attributes listed in this chapter are of interest to the application developer.

You can enforce file-wide constraints on files and fields with two Model 204 file models:

File model	**Action…**
Numeric Validation	Causes Model 204 to perform numeric data type validation on fields defined as FLOAT or BINARY.
1NF (First-Normal Form)	Ensures that the data within a file conforms to the rules for First-Normal Form.

Use the FILEMODL parameter to set a file model when creating a file.

Refer to the *Model 204 File Manager's Guide* for more information about the file model feature.

Field attribute descriptions

AT-MOST-ONE and REPEATABLE attributes

If a field is defined as having the AT-MOST-ONE attribute, Model 204 prevents multiple occurrences of that field in any given record. However, unlike fields with the OCCURS attribute, AT-MOST-ONE fields are not specifically preallocated.

If a field is *not* defined as AT-MOST-ONE, then it is REPEATABLE.

REPEATABLE is the default except for First-Normal Form files, where AT-MOST-ONE is required on all fields.

See the discussion "File model feature" on page 32-2.

AT-MOST-ONE versus UNIQUE attributes

Although the names of the UNIQUE and AT-MOST-ONE attributes sound similar, they have very different meanings:

- UNIQUE affects the value of the field
- AT-MOST-ONE affects the number of field occurrences per record

For example, if a Social Security field within an EMPLOYEE file is both UNIQUE and AT-MOST-ONE; the UNIQUE attribute ensures that the social security number for every employee is different, and AT-MOST-ONE ensures that each employee has only one social security number.

Both AT-MOST-ONE and UNIQUE are examples of "field level constraints."

FLOAT attribute

The FLOAT attribute is typically used if the field stores data in floating point representation. For detailed information about the handling of data for fields defined with the FLOAT attribute, refer to:

- "Equality retrievals" on page 4-16
- "Storing values in FLOAT or BINARY fields" on page 15-17

FOR EACH VALUE attribute

The FOR EACH VALUE attribute maintains a list of all the unique values created for a field. This list can be accessed by using a value loop statement, such as, FOR EACH VALUE, FOR *k* VALUES, or FIND ALL VALUES.

INVISIBLE attribute

The INVISIBLE attribute is required if the field is to be used in FILE RECORDS statements. It also can be used for other fields that have the KEY, NUMERIC RANGE, or ORDERED attribute. Such fields can be used for retrievals but cannot be used where the VISIBLE attribute is required.

An INVISIBLE field cannot be used in an arithmetic expression, as a key in a SORT statement, or in the AUDIT, COUNT OCCURRENCES, DELETE EACH, NOTE, PRINT, SET HEADER, and SET TRAILER statements. An INVISIBLE field also cannot be used in any form of the CHANGE or DELETE statement that is not followed by a fieldname = value pair.

Refer to the *Model 204 File Manager's Guide* for details on the use of INVISIBLE fields in 1NF (First-Normal Form) files.

KEY attribute

The KEY attribute specifies that if the field is to be used in FIND statement conditions of the form:

```
fieldname = value
```

the Table C index is searched, rather than the data in Table B. Thus, the selection of records based on KEY fields is substantially more efficient than selection based on NON-KEY fields.

In addition, either the KEY attribute or the ORDERED attribute is required if the field is to be used in FILE RECORDS statements.

LENGTH attribute

The LENGTH attribute specifies the preallocated length of a field occurrence in a record. See the discussion on preallocated fields in "Storing values in preallocated fields" on page 15-16.

NON-DEFERRABLE attribute

The NON-DEFERRABLE attribute causes a KEY, NUMERIC RANGE, or ORDERED retrieval field's index entries to be created immediately when the file is open in deferred update mode. All index entries are created immediately when a file is not in deferred update mode. The field cannot be located until its index entry has been created.

NUMERIC RANGE attribute

The NUMERIC RANGE attribute specifies that if the field is used in a retrieval statement that performs a numeric retrieval, the Table C index is searched, rather than the data in Table B. Numeric retrievals based on fields with the NUMERIC RANGE attribute are more efficient than numeric retrievals based on NON-RANGE fields. A NUMERIC RANGE field cannot be multiply occurring.

OCCURS attribute

The OCCURS attribute specifies the number of occurrences of a multiply occurring field that are preallocated in a record. See the discussion on preallocated fields in “Storing values in preallocated fields” on page 15-16.

ORDERED attribute

The ORDERED attribute specifies that the Ordered Index can be searched rather than the data in Table B for most types of retrieval. Thus, the selection of records based on ORDERED fields is substantially more efficient than selection based on NON-ORDERED fields.

A field with the ORDERED attribute also produces other efficiencies in User Language. For example, when the IN ORDER option is used on a FOR EACH RECORD statement, an internal sort is not required if the field is ORDERED.

The ORDERED attribute also allows a field to be used with any value loop statement, such as, FIND ALL VALUES, FOR EACH VALUE, and FOR k VALUES. In addition, either the ORDERED attribute or the KEY attribute is required if a field is to be used in FILE RECORDS statements.

UNIQUE attribute

The UNIQUE attribute automatically enforces a uniqueness constraint on fields; it ensures that a given field name = value pair occurs only in one record in a file.

VISIBLE attribute

The VISIBLE attribute is required if the field is to be used in NOTE, PRINT, or SORT statements, or in an arithmetic expression.

UPDATE attribute

The UPDATE attribute indicates the type of update method that is used when a field is changed.

- If UPDATE IN PLACE is specified, changing the value of a field occurrence does not change its position relative to other occurrences of the same field. The file manager usually specifies UPDATE IN PLACE.
- If UPDATE AT END is specified, a change in the value of a field occurrence is accomplished by deleting the existing occurrence and adding a new one following the others.

Refer to “UPDATE field attribute” on page 19-19 for detailed information on the handling of updates based on the type of update method specified.

33

DML statements in Parallel Query Option/204

In this chapter

- Parallel Query Option/204 DML
- Restricted commands and $functions

Parallel Query Option/204 DML

Parallel Query Option/204 data manipulation language (DML) consists of most, but not all, User Language statements and conditions. This chapter lists the statements and conditions that comprise Parallel Query Option/204 DML, and discusses certain restrictions on the use of some User Language commands, statements, and functions.

Listed below, in separate sections, are the Parallel Query Option/204 retrieval statements, retrieval conditions, and update statements. A discussion of IN clauses and field names in DML statements follows the listings.

DML statements and retrieval conditions

Retrieval statements

```
CLEAR LIST
COMMIT RELEASE
COUNT OCCURRENCES OF
COUNT RECORDS
DECLARE LIST
FIND RECORDS
FIND ALL RECORDS
FIND ALL VALUES
FIND AND PRINT COUNT
FOR EACH OCCURRENCE
FOR EACH RECORD
FOR EACH VALUE
FOR k RECORDS
FOR k VALUES
FOR RECORD NUMBER
NOTE fieldname
OPEN
OPENC
PLACE RECORD ON LIST
PLACE RECORDS ON LIST
PRINT ALL INFORMATION
RELEASE ALL RECORDS
RELEASE RECORDS
REMOVE RECORD FROM LIST
REMOVE RECORDS FROM LIST
SORT RECORD KEYS
SORT RECORDS
SORT VALUES
```

Retrieval conditions

```
FIND$
FILE$
LIST$
LOCATION$
```

```
POINT$
SFGE$
SFL$
```

Update statements

```
ADD
BACKOUT
CHANGE
COMMIT
DELETE
DELETE EACH
DELETE RECORD
DELETE RECORDS
FILE RECORDS
INSERT
STORE RECORD
UPDATE RECORD
```

Using IN clauses

A remote file specification (using either the AT clause or a file synonym) is syntactically valid in the IN FILE clause of supported DML statements in remote context.

Also, you can use the following IN clause variations to refer to a remote file if you are in scattered group context:

```
IN GROUP groupname MEMBER
IN $CURFILE
IN $UPDATE
```

Using field names in expressions

Fields in remote context can be used in the same ways as fields in local context including the VALUE IN phrase, which refers to a label on a FOR EACH RECORD IN ORDER statement.

Restricted commands and $functions

The Model 204 commands, User Language statements, and $functions that are not supported in remote context for Parallel Query Option/204 processing are listed in this section.

Restricted Model 204 commands

Some of the Model 204 commands that specify a file name, or that operate on a default file or on a file specified in an IN clause, are not supported in remote context for Parallel Query Option/204 processing.

The following commands cannot be used in reference to a remote file. See Table 16-1 on page 16-5 for a description of the commands that can be used to reference files in remote context:

ASSIGN (specifying a procedure alias)
BROADCAST FILE
CREATE FILE
DEASSIGN
DECREASE
DEFINE FIELD
DELETE FIELD
DELETE PROCEDURE
DESECURE
DESECURE PROCEDURE
DISPLAY FILE
DISPLAY LIST
DISPLAY PROCEDURE
DUMP
EDIT
ENQCTL
FILELOAD
FLOD
INCLUDE
INCREASE
INITIALIZE
PROCEDURE
REDEFINE FIELD
REGENERATE
RENAME FIELD
RENAME PROCEDURE
REORGANIZE
RESET (file parameters)
RESTORE
SECURE
SECURE PROCEDURE
TABLEB
TABLEC

TRANSFORM
Z

Restricted $functions

The following $functions cannot be used in reference to a remote file:

$LSTPROC
$RDPROC

If either of these $functions is used in reference to a remote file, Model 204 displays an error message.

A

Obsolete Features

In this appendix

- Statement numbers
- $DSCR function
- $TCAMFHP function

Statement numbers

This appendix describes certain features of User Language which, while still supported, are generally considered obsolete. They have been superseded by other, more efficient features or techniques. CCA does not recommend using these features in newly written requests and procedures.

CCA strongly recommends that you use statement labels (see “Statement labels” on page 30-2) rather than statement numbers. Although statement numbers are still supported, this section is included primarily as documentation for applications developed using earlier releases of Model 204.

FOPT parameter

The setting of the FOPT (File Options) parameter determines whether procedures within a file must be labeled or numbered. For more information on the FOPT parameter, refer to the *Model 204 Command Reference Manual*.

Rules for using statement numbers

The rules for specifying statement numbers in User Language statements are summarized below.

- A statement number is made up of the digits 0–9 and an optional period (.). The number must begin with a digit and cannot contain blanks, although blanks preceding or following it are optional.
- A statement number can start in any column up to but not including column INCCC.
- Any statement can be numbered, but some statements must be numbered. In particular, a statement must be numbered if:
  - It immediately follows a STORE RECORD or a FIND statement, in which case the statement number indicates the end of the preceding statement.
  - It is referred to by later statements. In most requests, COUNT RECORDS IN n, COUNT RECORD ON LIST m, NOTE, FIND, FOR EACH VALUE, and FOR EACH OCCURRENCE are referred to later and should therefore be numbered.
  - It is the first statement after a loop, in which case a number is needed to indicate the end of the loop. An END statement following a loop does not require a number.
  - It is the statement following a THEN or ELSE clause which is to be executed if the condition is false.
- The retrieval conditions of a FIND statement, the subexpressions of an IF statement, and the fields of a STORE RECORD statement must not be numbered, even if they start new lines.

- A statement number can have any number of parts, which are delimited by periods. Thus, 2.3 and 0.1 are two part numbers, and 0.05.4 is a three part number.
- A period (.) following a statement number is optional, but the periods separating the parts are required.

  For example:

  ```
  1.1
  ```

  and

  ```
  1.1.
  ```

  are both legal and are interpreted as the same number.

  This rule also applies when referring to statements by number. Thus:

  ```
  FOR EACH RECORD IN 1.
  ```

  and

  ```
  FOR EACH RECORD IN 1
  ```

  refer to the same statement.
- Any part of a statement number can be arbitrarily large.
- The number of parts in a statement number must be the same as the level on which the statement is nested. For example, the following sequence is illegal because 2.1.1 has three parts but is nested at the second level:

  ```
  2. FOR EACH RECORD IN 1
     2.1.1 PRINT ALL INFORMATION
  ```
- If a statement is not numbered, it is assigned a default level of nesting. The default level is the same as the level of the previous statement. If there is no previous statement, the default is the first level. If the previous statement starts a loop or a THEN, ELSE, or ELSEIF clause, or is a SUBROUTINE statement, the default level is one greater than the level of the previous statement.
- Any part of a statement number can have leading zeros. However, a number with leading zeros is not equivalent to one without. Thus 01 and 1 are both legal but not equivalent; the same is true of 1.1 and 1.01.
- Statements need not be numbered in sequential order. The statement number is just a label for the statement; its numerical value has no significance.

  The following, for instance, is a legal request:

  ```
  BEGIN
  3.    FIND ALL RECORDS FOR WHICH
        COMPANY = CCA
  1.    FOR EACH RECORD IN 3.
        3.1 PRINT ALL INFORMATION
  ```

```
2.6. SKIP 1 LINE
END
```

- Within a request, statement numbers should be unique. If not, Model 204 prints the message:

  ```
  M204.0223: STATEMENT LABEL MULTIPLY DEFINED
  ```

  This is a warning message; the request can still be run.

- A statement number followed by a label is allowed with or without a space between the number and the label. If a statement number is followed by a label and there is no space between them, User Language separates them into statement number and label. Although a statement number can be followed by a label, it is strongly recommended that you avoid placing statement numbers and labels on the same line.

Examples

The following request addresses an automated library card catalogue. Each record in the file contains information that would normally appear on a single card in the catalogue, such as AUTHOR, TITLE, SUBJECT, and CATALOGUE NUMBER.

```
BEGIN
 1. FIND ALL RECORDS FOR WHICH
       AUTHOR = PAULING
 2. FOR EACH RECORD IN 1
    2.1 NOTE SUBJECT
    2.2 FIND ALL RECORDS FOR WHICH
        SUBJECT = VALUE IN 2.1
    2.3 FOR EACH RECORD IN 2.2
        2.3.1 PRINT TITLE
 3. COUNT RECORDS IN 1
    PRINT COUNT IN 3
END
```

The following request uses a subroutine to validate full-screen input entries. A screen named DATA is defined with input areas named ITEM6, ITEM7, etc. Values entered during the READ SCREEN statement are validated one at a time by subroutine 900. Each screen item is assigned to the argument variable %A before the subroutine is called. If the item fails the validation, the subroutine sets %TAG to a nonzero value before returning. The request sets a tag on the screen for each incorrect item. The REREAD statement is issued if a screen item fails the validity tests.

```
BEGIN
SCREEN DATA
         .
         .
         .
END SCREEN
READ SCREEN DATA NO REREAD
      *VALIDATE DATA ITEMS
1.    %A = %DATA:ITEM6
      CALL 900
```

```
      IF %TAG THEN TAG %DATA:ITEM6
2.    %A = DATA:ITEM7
      CALL 900
         .
         .
         .
50.   IF $CHKTAG('DATA') THEN
      REREAD SCREEN DATA
      JUMP TO 1
51.   *PROCESS DATA
         .
         .
         .
      *SUBROUTINE TO PERFORM VALIDATION BY TABLE
LOOKUP
      * IN:       %A       VALUE
      * OUT:      %TAG     NONZERO IF INVALID
900.  SUBROUTINE
         .
         .
         .
END
```

$DSCR function

The $DSCR function interprets its character string argument as a field name. It returns a variable-length character string describing the specified field. New application development should incorporate “$FDEF” on page 27-51 that supplanted the $DSCR function.

The following discussion is provided to maintain existing code. Each letter in the returned string represents a particular field attribute. Attributes are listed in Table A-1.

Table A-1. $DSCR field attribute codes (file context)

Character	Attribute
A	ORDERED CHARACTER
C	CODED
D	DEFERRABLE
F	FRV
I	INVISIBLE
K	KEY
L	LEVEL
M	MANY-VALUED
N	ORDERED NUMERIC
O	OCCURS
P	UPDATE IN PLACE
Q	UNIQUE
R	NUMERIC RANGE
S	STRING
T	FLOAT
U	Undefined
W	AT-MOST-ONE

How $DSCR works

When $DSCR is invoked in file context, the returned string represents the description of the specified field in the current file.

Each letter that appears in the return value corresponds to an attribute in the field description. For example, if K is one of the letters in the returned string, then the field is KEY.

If a particular letter does not appear in the result of a $DSCR call, then either the corresponding attribute does not apply or the attribute's opposite is in effect. For example, if M (many-valued) does not appear, the field is NON-CODED and NON-FRV, in which case M does not apply, or it is FEW-VALUED. You can resolve this by looking for C and F in the returned string.

When $DSCR is invoked in group context, the returned string represents a composite description of the field in all of the files of the current group. Some of the letters imply that the corresponding field attribute is present in all of the files in the group; others imply that the attribute is present in some (at least one) file in the group. If the field specified as the $DSCR argument is not defined in the current file or group, $DSCR returns the character string "U" (undefined). Table A-2 lists the individual letters and their meanings in group context.

Table A-2. $DSCR field attribute codes (group context)

Character	Attribute
A	ORDERED CHARACTER in some
C	CODED in all
D	DEFERRABLE in some
F	FRV in some
I	INVISIBLE in all
K	KEY in all
L	LEVEL in some
M	MANY-VALUED in all
N	ORDERED NUMERIC in some
O	OCCURS in some
P	UPDATE IN PLACE in some
Q	UNIQUE in some
R	NUMERIC RANGE in all
S	STRING in all
T	FLOAT in some
W	AT-MOST-ONE in all
U	Undefined

Example

This request determines the attributes of a field and performs one of three types of searches, depending on the results of $DSCR.

```
BEGIN
```

```
             %A = $READ ('FIELD NAME')
             %B = $READ ('FIELD VALUE')
             %C = $DSCR (%A)
             IF %C EQ 'U' THEN
                PRINT 'ILLEGAL FIELD'
                JUMP TO STOP
             END IF
             *CHECK FOR NON-KEY
             IF $INDEX (%C, 'K') EQ 0 THEN
                JUMP TO NUM.RNG.CHK
             END IF
             FIND AND PRINT COUNT
                %%A = %B
             END FIND
             JUMP TO STOP

             *CHECK FOR NUMERIC RANGE
NUM.RNG.CHK: IF $INDEX (%C, 'R') EQ 0 THEN
                JUMP TO ALL.RECS
             END IF
             FIND AND PRINT COUNT
                %%A IS %B
             END FIND
             JUMP TO STOP
             *NEITHER KEY FOR NUMERIC RANGE
ALL.RECS:    FIND ALL RECORDS
             END FIND
             FOR EACH RECORD IN ALL.RECS
                IF %%A = %B THEN
                   PLACE RECORD ON LIST OK
                END IF
             END FOR
OKS:         COUNT RECORDS ON LIST OK
             PRINT COUNT IN OKS
STOP:        END
END
```

$TCAMFHP function

The $TCAMFHP function returns a string that contains the user's current TCAM fixed header prefix (FHP). The FHP returned is not formatted. The user must use $SUBSTR to extract the desired components of the FHP. $TCAMFHP takes no arguments. $TCAMFHP is meaningful only for users of TCAM 3270s (IODEV 21) which are no longer supported by CCA. If FHPs are not in use or the user does not have an IODEV of 21, a null string is returned.

Index