iSeries™

VisualAge® RPG Language Reference

Version 6.0 for Windows®

SC09-2451-06

iSeries™

VisualAge® RPG Language Reference

Version 6.0 for Windows®

SC09-2451-06

Note!
Before using this information and the product it supports, be sure to read the general information under “Notices” on page 747.

Seventh Edition (June 2005)

This edition applies to Version 6.0 of IBM WebSphere Development Studio Client for iSeries and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC09-2451-05 .

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send your comments to:

IBM Canada Ltd. Laboratory
Information Development
D1/817/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

You can also send your comments electronically to IBM. See “How to Send Your Comments” on page xii for a description of the methods.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About This Book

This book provides information about the RPG IV language as implemented using the VisualAge RPG compiler with the Windows® operating system.

This book contains:
- Language fundamentals, such as, the character set, symbolic names, reserved words, compiler directives, and indicators
- Data types and data formats
- Error and exception handling
- Subprocedures
- Specifications
- Built-in functions, expressions, and operation codes.

This book is for programmers who are familiar with the VisualAge RPG programming language.

This reference provides a detailed description of the VisualAge RPG language. It does not provide information on how to use the VisualAge RPG compiler or how to convert ILE RPG programs to VisualAge RPG programs. For more information on these topics, see *Programming with VisualAge RPG*, SC09-2449-05.

Before using this book, you should be familiar with the tasks for a VisualAge RPG application. Refer to *Programming with VisualAge RPG* and the online help.

Prerequisite and Related Information

Use the iSeries Information Center as your starting point for looking up iSeries and AS/400e technical information. You can access the Information Center in two ways:
- From the following Web site:

  `http://www.ibm.com/eserver/iseries/infocenter`
- From CD-ROMs that ship with your OS/400 order:

  iSeries Information Center, SK3T-4091-00. This package also includes the PDF versions of iSeries manuals, *iSeries Information Center: Supplemental Manuals*, SK3T-4092-00, which replaces the Softcopy Library CD-ROM.

The iSeries Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitions, clustering, Java ™ , TCP/IP, Web serving, and secured networks. It also includes links to related IBM® Redbooks and Internet links to other IBM Web sites such as the Technical Studio and the IBM home page.

The VisualAge RPG Library

The VisualAge RPG library contains the following publications:

Programming with VisualAge RPG

This book contains specific information about creating applications with VisualAge RPG. It describes the steps you have to follow at every stage of the application

development cycle, from design to packaging and distribution. Programming examples are included to clarify the concepts and the process of developing VisualAge RPG applications.

VisualAge RPG Parts Reference

This book provides information on the VisualAge RPG **parts**, **part attributes**, **part events**, and **event attributes**. It is a reference for anyone who is developing applications using VisualAge RPG.

VisualAge RPG Language Reference

This book provides information about the RPG IV language as implemented using the VisualAge RPG compiler. It contains:
- Language fundamentals such as the character set, symbolic names and reserved words, compiler directives, and indicators
- Data types and data formats
- Error and exception handling
- Specifications
- Built-in functions, expressions, and operation codes.

For an overview of the entire product, see *Getting Started with WebSphere Development Studio Client for iSeries*.

For a list of **related publications**, see the Bibliography at the end of this book.

You can also find the most current information about IBM WebSphere Development Studio Client for iSeries on the following online source:

The Development Studio Client Home Page
`ibm.com/software/ad/wdsc/`

How to Send Your Comments

Your feedback is important in helping us to provide the highest quality information possible. IBM welcomes any comments about this book or any other iSeries documentation.
- If you prefer to send comments by mail, use the following address:

  IBM Canada Ltd. Laboratory
  Information Development
  D1/817/8200/MKM
  8200 Warden Avenue
  Markham, Ontario, Canada L6G 1C7
- If you prefer to send comments electronically, use this e-mail address:

  toreador@ca.ibm.com
- If you prefer to send comments by fax, use this number:

  1-845-491-7727

Be sure to include the following:
- The name of the book
- The publication number of the book
- The page number or topic to which your comment applies.

Accessing Online Information

VisualAge RPG contains a variety of online books and online help. You can access the help while you are using the product, and can view the books either while you are using the product, or independently.

Using Online Books

To view an online book, either:

- Select the name of the book from the **Help** pull-down menu of the VisualAge RPG GUI Designer or the editor window.
- Access the books from the **Start** menu. Select **Programs → IBM WebSphere Development Studio Client for iSeries**. Then select **Documentation**.

Publications in PDF Format

VisualAge RPG publications are available in Portable Document Format (PDF) from the iSeries Information Center at URL `http://www.ibm.com/eserver/iseries/infocenter` .

Note: You need the Adobe Acrobat Reader, Version 3.01 or later for Windows, to view the PDF format of our publications on the workstation. If your location does not have the reader, you can download a copy from the Adobe Systems Web site (http://www.adobe.com).

The following VisualAge RPG publications are available in PDF format:

- *Programming with VisualAge RPG*
- *VisualAge RPG Parts Reference*
- *VisualAge RPG Language Reference*

For information on the product, see *Getting Started with WebSphere Development Studio Client for iSeries*, SC09-2625-06.

Using Online Help

Online help is available for all areas of VisualAge RPG. To get help for a particular window, dialog box, or properties notebook, select the **Help** push button (when available).

Note: To view help that is in HTML format, your workstation must have a frames-capable Web browser, such as Netscape Navigator 4.04 or higher, or Microsoft® Internet Explorer 4.01 or higher. (Recommended browser is Netscape Navigator 4.6 or Internet Explorer 5.0)

Using context-sensitive help

To receive context-sensitive help at any time, press F1. The help that appears is specific to the area of the interface that has input focus. Input focus can be on menu items, windows, dialog boxes, and properties notebooks, or on specific parts of these.

For context-sensitive help on dialog boxes, click on the question mark (when available) in the top right-hand corner of the window. A question mark will appear beside the mouse arrow. Click on a word or field and help information on that specific field will be displayed.

Using language-sensitive help

To receive language sensitive help, press F1 in an edit window. If the cursor is on an operation code, you receive help for that operation code; otherwise, you receive help for the current specification.

What's New in Version 6.0

This publication includes information from previous release Readmes and other technical corrections. Changes are noted by a vertical bar (|).

Changes include:

- New built-in function %SUBARR (assign to, sort, or return, a subarray).
- Direct conversion of date/time/timestamp to numeric, using %DEC.
- Second parameter for %TRIM, %TRIMR and %TRIML indicating what characters to trim.
- New prototype option OPTIONS(*TRIM) to pass a trimmed parameter.
- Relaxation of the rules for using a result data structure for I/O to externally-described files and record formats.

Part 1. Introduction to the VisualAge RPG Language

This section describes some of the basic elements of the VisualAge® RPG (VARPG) language such as:

- Character set
- Symbolic names and reserved words
- Compiler directives
- Indicators
- Subprocedures

Chapter 1. Symbolic Names and Reserved Words

The valid character set for the VisualAge RPG language consists of the following:

letters A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Lowercase letters in symbolic names can be used, however, they are translated to uppercase during compilation.

numbers 0 1 2 3 4 5 6 7 8 9

characters + − * , . ' & / $ # : @ _ > < = ( ) %

The blank character

Symbolic Names

A symbolic name uniquely identifies specific data in a program or procedure. Its purpose is to allow you to access that data. The following rules apply to all symbolic names:

- The first character of the name must be alphabetic, $, #, or @
- The remaining characters must be alphabetic, numeric, or the underscore (_)
- The name must be left-adjusted in the entry on the specification form except in fields which allow the name to float (definition specification, keyword fields, and the extended-factor 2 field)
- A symbolic name cannot be a reserved word
- A symbolic name can be from 1 to 4096 characters. The practical limits are determined by the size of the entry used for defining the name. A name that is up to 15 characters can be specified in the Name entry of the definition or procedure specification. For names longer than 15 characters, use a continuation specification.
- A symbolic name must be unique within the procedure in which it is defined.

Table 1 lists symbolic names and any additional restrictions.

Table 1. Restrictions for Symbolic Names

Arrays	An array name in a standalone field cannot begin with the letters TAB. Array names may begin with TAB if they are either prototyped parameters or data structures defined with the DIM keyword.
Conditional compilation names	Symbolic names used for conditional compilation have no relationship to other symbolic names. Names for conditional compilation can be up to 50 characters long.
Data structures	A data structure name can only be defined once.
Exception output records	The same EXCEPT name can be assigned to more than one exception output record.
Fields	• A field name can be defined more than once if each definition using that name has the same data type, the same length, and the same number of decimal positions. All definitions using the same name refer to a single field (that is, the same area in storage). However, it can be defined only once on the definition specification. • A field can be defined as a data structure subfield only once unless the data structure is qualified (defined with QUALIFIED or LIKEDS). In this case, when the subfield is used, it must be qualified (specified in the form *dsname*.*subfieldname*). • A subfield name cannot be specified as the result field on an *ENTRY PLIST parameter. The VisualAge RPG compiler creates global fields for static text and entry field parts with the same name as the part. Any explicit definitions of these field names in your source must match.
Key field lists	There are no additional restrictions to key field list (KLIST) names.
Labels	There are no additional restrictions to label names.
Named constants	There are no additional restrictions to named constants.
Parameter lists	There are no additional restrictions to parameter list (PLIST) names.
Prototype names	There are no additional restrictions to prototype names.
Record names	A record name can exist in only one file in the program.
Subroutines	See "BEGACT (Begin Action Subroutine)" on page 508 for a description of action subroutine names and "BEGSR (Begin User Subroutine)" on page 511 for a description of user subroutine names.
Tables	A table name can contain from 3 to 10 characters, must begin with the characters TAB, and cannot be defined in a subprocedure.
Windows	Window names defined in the component's GUI definition are reserved as symbolic names in the program, even within procedures.

Words with Special Functions and Reserved Words

The following is a summary of words with special functions.

Built-in Function Special Words

The *ALL and *NULL special words are used with built-in functions. For more information on built-in functions, see “Built-In Functions (Alphabetically)” on page 405.

Date and Time Special Words

The following special words are used with Date, Time, and Timestamp fields:

*CDMY	*CMDY	*CYMD
*CYMD0	*DMY	*ISO
*LONGJUL	*MDY	*EUR
*JIS	*USA	*HMS
*JUL	*YMD	

For more information on date formats, see “DATFMT(fmt{separator})” on page 227.

Expressions

The NOT special word can be used with expressions. For more information on expressions, see Chapter 24, “Expressions,” on page 381.

File Positioning Special Words

The *START and *END special words can be used to position in a file. For more information on file positioning, see “File Positioning” on page 6.

Implied Literals

Figurative constants are implied literals that allow specifications without referring to length. For more information on figurative constants, see “Figurative Constants” on page 153.

*ALLX’x1..’	*DARKGREEN	*OFF
*ALLG’K1K2’	*DARKGRAY	*ON
*ALL’X..’	*DARKPINK	*OK
*ABORT	*DARKRED	*PALEGRAY
*BLACK	*ENTER	*PINK
*BLANK	*GREEN	*RED
*BLANKS	*HALT	*RETRY
*BLUE	*HIVAL	*WARN
*BROWN	*INFO	*WHITE
*CANCEL	*IGNORE	*YELLOW
*CYAN	*LOVAL	*YESBUTTON
*DARKBLUE	*NOBUTTON	*ZERO
*DARKCYAN	*NULL	*ZEROS

Indicator Reserved Words

The *IN and *INxx reserved words allow indicators to be referred to as data. For more information, see “Indicators Referred to as Data” on page 26.

Job Date Reserved Words

The following reserved words allow you to access the job date, or a portion of it. For more information, see “User Date Special Words” on page 8.

UDATE	*DATE
UMONTH	*MONTH
UYEAR	*YEAR
UDAY	*DAY

Page Numbering Reserved Words

The PAGE and PAGE1-PAGE7 reserved words can be used for numbering the pages of a report or to sequentially number output fields. For more information, see “PAGE, PAGE1-PAGE7 Reserved Words” on page 7.

Parameter Passing Special Words

The *OMIT, *RIGHTADJ, *STRING, *TRIM, and *VARSIZE special words are used for parameter passing.

Placement of Fields

*PLACE allows repetitive placement of fields in an output record. For more information, see “*PLACE” on page 327.

Writing all Fields

*ALL allows all fields that are defined for an externally described file to be written on output. For a more information on figurative constants, see “Rules for Figurative Constants” on page 154.

File Positioning

*START and *END change the position of an OS/400™ database file.

If the file is a non-keyed file, *START and *END position to the start and end of the file, respectively. If the file is a keyed file, *START and *END position to the start and end of the keyed access path, respectively.

PAGE, PAGE1-PAGE7 Reserved Words

PAGE is used to number the pages of a report, to serially number the output records in a file, or to sequentially number output fields. It does not cause a page eject.

The eight possible PAGE fields (PAGE, PAGE1, PAGE2, PAGE3, PAGE4, PAGE5, PAGE6, and PAGE7) may be used to number different types of output pages or to number pages for different printer files.

PAGE fields can be specified in positions 30 through 43 of the output specifications or in the input or calculation specifications.

The following rules apply to the PAGE fields:
- Page numbering, unless otherwise specified, starts with 1
- For each new page, 1 is automatically added
- PAGE fields can be any length
- PAGE fields must have zero decimal positions
- When a PAGE field is only specified in the output specifications, it is treated as a four digit, numeric field with zero positions.

You can use the PAGE words in a variety of ways:
- To start at a page number other than 1, set the value of the PAGE field to one less than the desired starting page.
- To restart page numbering at any point in a job:
  - Specify blank after (position 45 of the output specifications)
  - Specify the PAGE field as the result field of an operation in the calculation specifications
  - Specify an output indicator in the output field (see Figure 2). When the output indicator is set on, the PAGE field is reset to 1. Output indicators cannot be used to control printing of a PAGE field because a PAGE field is always written.
  - Specify the PAGE field as an input field (see Figure 1).

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC..............................
I.......................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
IINPUT         50   1  CP
I                                  2    5 0PAGE
```

Figure 1. Page Record Description

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
OFilename++EF..N01N02N03Excnam++++B++A++Sb+Sa+..........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
O*  When indicator 15 is on, the PAGE field is set to zero and 1 is
O*  added before the field is printed.  When indicator 15 is off, 1
O*  is added to the contents of the PAGE field before it is printed.
OPRINT     E    99                        01
O               15      PAGE          1     75
```

Figure 2. Resetting the PAGE Fields to Zero

User Date Special Words

A date for a program can be specified at runtime by using a user date special word. The user date special words are: UDATE, *DATE, UMONTH, *MONTH, UDAY, *DAY, UYEAR, and *YEAR.

The user date special words access the job date that is specified in the job description. The user date can be written out as output.

The user date special words are set when the application starts running. They are not updated when the program runs over midnight or when the job date changes. Use the TIME operation to obtain the time and date while the program is running. For more information on the TIME operation, see "TIME (Time of Day)" on page 707.

Use the DATEDIT keyword on the control specification to specify the date formats of UDATE and *DATE:

DATEDIT	UDATE format	*DATE format
*MDY	*MDY	*USA (mmddyyyy)
*DMY	*DMY	*EUR (ddmmyyyy)
*YMD	*YMD	*ISO (yyyymmdd)

If this keyword is not specified, the default is *MDY.

The following restrictions apply to user date fields:
- User date fields are numeric fields, not date type fields.
- User date fields cannot be modified. This means that they cannot be used:
  - In the result field of calculations
  - As factor 1 of PARM operations
  - As factor 2 index of LOOKUP operations
  - With blank after in output specifications
  - As input fields
- The user date fields UMONTH, *MONTH, UDAY, *DAY, UYEAR, and *YEAR cannot be edited by the Y edit code in position 44 of the output specifications.

You can use the user date words in a variety of ways:

Operation codes using numeric fields	The user date special words can be used in factor 1 or factor 2 of the calculation specifications for operation codes that use numeric fields.
Editing UDATE and *DATE	UDATE and *DATE can be edited when they are written if the & edit code is specified in position 44 of the output specification. The DATEDIT keyword on the control specification determines the format and the separator character to be inserted.
Printing 2-position date fields	To print a 2-position date field, specify UMONTH, *MONTH UDAY, *DAY, and UYEAR on the output specifications.
Printing 4-position date fields	To print a 4-position date field, specify UMONTH, *MONTH UDAY, *DAY, and UYEAR on the output specifications.
Printing 6-position date fields	To print a 6-position date field, specify UDATE on the output specifications. Three different date formats can be used: Month/day/year, Year/month/day, Day/month/year. The DATEDIT keyword on the control specification determines the format.
Printing 8-position date fields	To print a 8-position date field, specify *DATE on the output specifications. The year is four digits. Three different date formats can be used: Month/day/year, Year/month/day, Day/month/year. The DATEDIT keyword on the control specification determines the format.
Printing the day	To print only the day, specify UDAY or *DAY on the output specifications.
Printing the month	To print only the month, specify UMONTH or *MONTH on the output specifications.
Printing the year	To print only the year, specify YEAR or *YEAR on the output specifications.

Chapter 2. Compiler Directives

The compiler directive statements /FREE... /END-FREE denote a free-form calculation specification block. The compiler directives /TITLE, /EJECT, /SPACE, /COPY and /INCLUDE and /INCLUDE allow you to specify heading information for the compiler listing, to control the spacing of the compiler listing, and to insert records from other file members during a compile. The conditional compiler directive statements /DEFINE, /UNDEFINE, /IF, /ELSEIF, /ELSE, and /EOF allow you to select or omit source records. The compiler directive statements must precede any compile-time arrays or table records.

/FREE... /END-FREE (Positions 7-11)

Positions	Entry
7-11	/FREE or /END-FREE
12-80	Blank

The /FREE compiler directive specifies the beginning of a free-form calculation specifications block. /END-FREE specifies the end of the block. Positions 12 through 80 must be blank. The remaining positions may be used for comments. See “Free-Form Syntax” on page 318 for information on using free-form statements.

/COPY or /INCLUDE)

The /COPY and /INCLUDE directives have the same purpose and the same syntax. You can freely choose which directive to use.

The /COPY and /INCLUDE compiler directives cause records from other files to be inserted, at the point where the directive occurs, within the file being compiled. This file can exist on your workstation or on an iSeries server. The inserted records can contain any valid specification, including /COPY and /INCLUDE, up to the maximum nesting depth specified by the COPYNEST keyword (32 when not specified).

To facilitate application maintenance, you may want to place the prototypes of exported procedures in a separate source member. If you do, be sure to place a /COPY or /INCLUDE directive for that member in both the module containing the exported procedure and any modules that contain calls to the exported procedure.

The copy directive is not printed on the compiler listing, but is replaced by the contents of the specified file. All copied files appear in the COPY member table of the compiler listing.

/COPY members are considered fixed-form by default, even if the /COPY directive is coded within a free-form group. If the /COPY member will contain free-form specifications, these must be delimited with /FREE and /END-FREE directives.

Copying Files from an iSeries Server

To copy files from an iSeries server, enter the /COPY statement as follows:

- /COPY or /INCLUDE followed by exactly one space
- *REMOTE followed by exactly one space
- The location of the member to be copied (merged). The format is: libraryname/filename,membername
  - A member name must be specified.
  - If a file name is not specified, QRPGLESRC is the default.
  - A comma separates filename and membername. The comma must be included.
  - If a library is not specified, the library list is searched for the file. All occurrences of the specified source file in the library list are searched for the member until it is located or the search is complete.
  - If a library is specified, a file name must also be specified.
- Optionally, at least one space and a comment.

The following are examples of the /COPY statement for copying OS/400 files:

- To copy the member MBR1 in the source file QRPGLESRC, enter the following statement. Note that the current library list is used to search for file QRPGLESRC:

  ```
  C/COPY *REMOTE MBR1
  ```

- To copy the member MBR1 in the source file SRCFIL, enter the following statement. Note that the current library list is used to search for file SRCFIL:

  ```
  I/COPY *REMOTE SRCFIL,MBR1
  ```

- To copy the member MBR1 in the source file SRCFIL in the library SRCLIB, enter the following statement:

  ```
  O/COPY *REMOTE SRCLIB/SRCFIL,MBR1
  ```

- To copy the member ″mbr1″ in file ″srcfil″ in library ″srclib″, enter the following statement:

  ```
  O/COPY *REMOTE "srclib"/"srcfil","mbr1"
  ```

Copying Files from a Workstation

To copy files from a local workstation, enter the /COPY statement as follows:

- /COPY or /INCLUDE followed by exactly one space
- The location of the member to be copied. The format is: Drive:\pathname\member.CPY

  Drive and path are optional.
- Optionally, at least one space and a comment.

The following example illustrates the /COPY statement for copying local files:

```
O/COPY D:\PROJECT1\INCLUDES\TOOLS1.CPY
```

Nested /COPY or /INCLUDE

Nesting of /COPY and /INCLUDE directives is allowed. A /COPY or /INCLUDE member may contain one or more /COPY or /INCLUDE directives (which in turn may contain further /COPY or /INCLUDE directives and so on). The maximum depth to which nesting can occur can be set using the COPYNEST control specification keyword. The default maximum depth is 32.

You must ensure that your nested /COPY or /INCLUDE files do not include each other infinitely. Use conditional compilation directives at the beginning of your /COPY or /INCLUDE files to prevent the source lines from being used more than once.

Conditional Compilation Directives

The conditional compilation directive statements allow you to conditionally include or exclude sections of source code from the compilation.

- Condition-names can be added or removed from a list of currently-defined conditions using the defining condition directives /DEFINE and /UNDEFINE.
- Conditional expressions DEFINED(condition-name) and NOT DEFINED(condition-name) are used within testing condition /IF groups.
- Testing conditional directives, /IF, /ELSEIF, /ELSE, and /ENDIF, control which source lines are to be read by the compiler.
- The /EOF directive tells the compiler to ignore the rest of the source lines in the current source member.

Defining Conditions

Condition-names can be added to or removed from a list of currently-defined conditions using the defining condition directives /DEFINE and /UNDEFINE.

/DEFINE (Positions 7-13)

The /DEFINE compiler directive defines conditions for conditional compilation. The entries in the condition-name area are free-format (do not have to be left justified). The following entries are used for /DEFINE:

Positions	Entry
7 - 13	/DEFINE
14	Blank
15 - 80	condition-name
81 - 100	Comments

The /DEFINE directive adds a condition-name to the list of currently-defined conditions. A subsequent /IF DEFINED(condition-name) would be true. A subsequent /IF NOT DEFINED(condition-name) would be false.

/UNDEFINE (Positions 7-15)

Use the /UNDEFINE directive to indicate that a condition is no longer defined. The entries in the condition-name area are free-format (do not have to be left justified).

Positions	Entry
7 - 15	/UNDEFINE
16	Blank
17 - 80	condition-name
81 - 100	Comments

The /UNDEFINE directive removes a condition-name from the list of currently-defined conditions. A subsequent /IF DEFINED(condition-name) would be false. A subsequent /IF NOT DEFINED(condition-name) would be true.

Note: Any conditions specified on the DEFINE parameter will be considered to be defined when processing /IF and /ELSEIF directives. These conditions can be removed using the /UNDEFINE directive.

Predefined Conditions

Several conditions are defined for you by the RPG compiler. These conditions cannot be used with /DEFINE or /UNDEFINE. They can only be used with /IF and /ELSEIF.

Conditions Relating to the Compiler Target

COMPILE_WINDOWS
This condition is defined if your program is being compiled to produce a Windows native program. (EXE or DLL object.)

COMPILE_JAVA
This condition is defined if your program is being compiled to run in Java.

Conditional Expressions

A conditional expression has one of the following forms:
- DEFINED(condition-name)
- NOT DEFINED(condition-name)

The condition expression is free-format but cannot be continued to the next line.

Testing Conditions

Conditions are tested using /IF groups, consisting of an /IF directive, followed by zero or more /ELSEIF directives, followed optionally by an /ELSE directive, followed by an /ENDIF directive.

Any source lines except compile-time data, are valid between the directives of an /IF group. This includes nested /IF groups.

Note: There is no practical limit to the nesting level of /IF groups.

/IF Condition-Expression (Positions 7-9)

The /IF compiler directive is used to test a condition expression for conditional compilation. The following entries are used for /IF:

Positions	Entry
7 - 9	/IF
10	Blank
11 - 80	Condition expression
81 - 100	Comments

If the condition expression is true, source lines following the /IF directive are selected to be read by the compiler. Otherwise, lines are excluded until the next /ELSEIF, /ELSE or /ENDIF in the same /IF group.

/ELSEIF Condition-Expression (Positions 7-13)

The /ELSEIF compiler directive is used to test a condition expression within an /IF or /ELSEIF group. The following entries are used for /ELSEIF:

Positions	Entry
7 - 13	/ELSEIF
14	Blank

15 - 80	Condition expression
81 - 100	Comments

If the previous /IF or /ELSEIF was not satisfied, and the condition expression is true, then source lines following the /ELSEIF directive are selected to be read. Otherwise, lines are excluded until the next /ELSEIF, /ELSE or /ENDIF in the same /IF group is encountered.

/ELSE (Positions 7-11)

The /ELSE compiler directive is used to unconditionally select source lines to be read following a failed /IF or /ELSEIF test. The following entries are used for /ELSE:

Positions	Entry
7 - 11	/ELSE
12 - 80	Blank
81 - 100	Comments

If the previous /IF or /ELSEIF was not satisfied, source lines are selected until the next /ENDIF.

If the previous /IF or /ELSEIF was satisfied, source lines are excluded until the next /ENDIF.

/ENDIF (Positions 7-12)

The /ENDIF compiler directive is used to end the most recent /IF, /ELSEIF, or /ELSE group. The following entries are used for /ENDIF:

Positions	Entry
7 - 12	/ENDIF
13 - 80	Blank
81 - 100	Comments

Following the /ENDIF directive, if the matching /IF directive was a selected line, lines are unconditionally selected. Otherwise, the entire /IF group was not selected, so lines continue to be not selected.

Rules for Testing Conditions

- /ELSEIF, and /ELSE are not valid outside an /IF group.
- An /IF group can contain at most one /ELSE directive. An /ELSEIF directive cannot follow an /ELSE directive.
- /ENDIF is not valid outside an /IF, /ELSEIF or /ELSE group.
- Every /IF must be matched by a subsequent /ENDIF.
- All the directives associated with any one /IF group must be in the same source file. It is not valid to have /IF in one file and the matching /ENDIF in another, even if the second file is in a nested /COPY. However, a complete /IF group can be in a nested /COPY.

The /EOF Directive

The /EOF directive tells the compiler to ignore the rest of the source lines in the current source member.

/EOF (Positions 7-10)

The /EOF compiler directive is used to indicate that the compiler should consider that end-of-file has been reached for the current source file. The following entries are used for /EOF:

Positions	Entry
7 - 10	/EOF
11 - 80	Blank
81 - 100	Comments

/EOF will end any active /IF group that became active during the reading of the current source member. If the /EOF was in a /COPY file, then any conditions that were active when the /COPY directive was read will still be active.

Note: If excluded lines are being printed on the listing, the source lines will continue to be read and listed after /EOF, but the content of the lines will be completely ignored by the compiler. No diagnostic messages will ever be issued after /EOF.

Using the /EOF directive will enhance compile-time performance when an entire /COPY member is to be used only once, but may be copied in multiple times. (This is not true if excluded lines are being printed).

/EJECT (Positions 7-12)

Use the compiler directive /EJECT to begin a new page on the compiler listing.

Note: /EJECT is not printed on the compiler listing, but is replaced by a new page. If the compiler listing is already at the top of a new page, a new page is not printed on the compiler listing.

To specify a new page, enter the /EJECT statement as follows:

Positions	Entry
7-12	/EJECT
13-49	Blank
50-100	Comments

/SPACE (Positions 7-12)

Use the compiler directive /SPACE to control line spacing within the source section of the compiler listing.

Note: /SPACE is not printed on the compiler listing, but is replaced by the specified line spacing. The line spacing caused by /SPACE is in addition to the two lines that are skipped between specification types.

To specify heading information, enter the /SPACE statement as follows:

Positions	Entry
7-12	/SPACE
13	Blank
14-16	A positive integer value from 1 through 112 that defines the number of lines to space. If a number greater than 112 is specified, 112 is used as the /SPACE value. If the number is greater than the number of lines remaining on the current page, subsequent specifications begin at the top of the next page.
17-49	Blank
50-100	Comments

/TITLE (Positions 7-12)

Use the compiler directive /TITLE to specify heading information (such as security classification or titles). This title information appears at the top of each page of the compiler listing.

A program can contain more than one /TITLE statement. Each /TITLE statement provides heading information for the compiler listing until another /TITLE statement is encountered. A /TITLE statement must be the first specification encountered in order to print information on the first page of the compiler listing. The information specified by the /TITLE statement is printed in addition to compiler heading information.

Note: /TITLE is not printed on the compiler listing, but is replaced by the heading information. The /TITLE statement causes a skip to the next page before the title is printed.

/FREE... /END-FREE (Positions 7-12)

To specify heading information, enter the /TITLE statement as follows:

Positions	Entry
7-12	/TITLE
13	Blank
14-100	Title information

Chapter 3. Indicators

An indicator is a one byte character field which contains either '1' (on) or '0' (off). Indicators are generally used to indicate the result of an operation or to condition the processing of an operation.

Indicators are defined either by an entry on the specification. The positions on the specification where an indicator is defined determine how the indicator is used. An indicator that has been defined can then be used to condition calculation and output operations.

The indicator format can be specified on the definition specifications to define indicator variables. For a description of how to define character data in the indicator format, see "Character Data Type" on page 110 and "Position 40 (Internal Data Type)" on page 263.

The state of most indicators can be changed by calculation operations. All indicators can be set on with the SETON operation code and set off with the SETOFF operation code.

This section describes:
- Indicators defined on the VisualAge RPG specifications (record identifying indicators, field indicators, resulting indicators)
- The Last Record Indicator (LR)
- Assigning field record relation indicators
- Conditioning calculations
- Using indicators in expressions
- Conditioning output
- Indicators referred to as data.

Indicators Defined on the Specifications

The following indicators can be defined on the specifications:
- Record identifying indicator (positions 21 and 22 of the input specifications)
- Field indicator (positions 69 through 74 of the input specifications)
- Resulting indicator (positions 71 through 76 of the calculation specifications)
- *IN array, *IN(xx) array element or *INxx field.

The defined indicator can then be used to condition operations in the program.

Record Identifying Indicators

A record identifying indicator is defined by an entry in positions 21 and 22 of the input specifications and is set on when the corresponding record type is selected for processing. That indicator can then be used to condition certain calculation and output operations. Record identifying indicators do not have to be assigned in any particular order.

The record identifying indicators are 01-99 and LR.

For an externally described file, a record identifying indicator is optional. If it is specified, it follows the same rules as for a program described file.

When a record type is selected for processing, the corresponding record identifying indicator is set on. All other record identifying indicators are off except when a file operation code is used to retrieve records from a file. The record identifying indicator is set on after the record is selected, but before the input fields are moved to the input area. Indicators can be set off at any time.

If file operation code is used on the calculation specifications to retrieve a record, the record identifying indicator is set on as soon as the record is retrieved from the file. It is possible to have several record identifying indicators for the same file, as well as record-not-found indicators, set on concurrently if several operations are issued to the same file.

Rules for Assigning Record Identifying Indicators

The following rules apply when assigning record identifying indicators to records in a program described file:

- The same indicator can be assigned to two or more different record types if the same operation is to be processed on all record types. To do this, specify the record identifying indicator in positions 21 and 22, and specify the record identification codes for the various record types in an OR relationship.
- A record identifying indicator can be associated with an AND relationship, but it must appear on the first line of the group. Record identifying indicators cannot be specified on AND lines.
- An undefined record (a record in a program described file that was not described by a record identification code in positions 23 through 46) causes the program to halt.
- A record identifying indicator can be specified as a record identifying indicator for another record type, as a field indicator, or as a resulting indicator. No diagnostic message is issued, but this use of indicators may cause erroneous results.

The following rules apply when assigning record identifying indicators to records in an externally described file:

- AND/OR relationships cannot be used with record format names; however, the same record identifying indicator can be assigned to more than one record.
- The record format name, rather than the file name, must be specified in positions 7 through 16.

Field Indicators

A field indicator is defined by an entry in positions 69 and 70, 71 and 72, or 73 and 74 of the input specifications. The field indicators are the general indicators 01-99.

A field indicator can be used to determine if the specified field or array element is greater than zero, less than zero, zero, or blank:

- Positions 69 through 72 are valid for numeric fields
- Positions 73 and 74 are valid for numeric or character fields
- An indicator specified in positions 69 and 70 is set on when the numeric input field is greater than zero
- An indicator specified in positions 71 and 72 is set on when the numeric input field is less than zero
- An indicator specified in positions 73 and 74 is set on when the numeric input field is zero or when the character input field is blank.

The field indicator can then be used to condition calculation or output operations.

A field indicator is set on when the data for the field or array element is extracted from the record and the condition it represents is present in the input record. This

field indicator remains on until another record of the same type is read and the condition it represents is not present in the input record, or until the indicator is set off as the result of a calculation.

Rules for Assigning Field Indicators

The following rules apply when assigning field indicators:

- Indicators for plus, minus, zero, or blank are set off at the beginning of the program. They are not set on until the condition (plus, minus, zero, or blank) is satisfied by the field being tested on the record just read.
- Field indicators cannot be used with entire arrays. However,an entry can be madefor an array element. Field indicators are allowed for null-capable fields only if the **User control** or **ALWNULL(*USRCTL)** option is used. See "Database Null Value Support" on page 137 for information on null value support.
- A numeric input field can be assigned two or three field indicators. However, only the indicator that signals the result of the test on that field is set on; the others are set off.
- If the same field indicator is assigned to fields in different record types, its state (on or off) is always based on the last record type selected.
- When different field indicators are assigned to fields in different record types, a field indicator remains on until another record of that type is read. Similarly, a field indicator assigned to more than one field within a single record type always reflects the status of the last field defined.
- The same field indicator can be specified as a field indicator on another input specification, as a resulting indicator, as a record identifying indicator, or as a field record relation indicator. No diagnostic message is issued, but this use of indicators could cause erroneous results.
- If the same indicator is specified in all three positions, the indicator is always set on when the record containing this field is selected.

Resulting Indicators

Resulting indicators are used by calculation specifications in the traditional format (C specifications). They are not used by free-form calculation specifications. For most operation codes, in either traditional format or free-form, you can use built-in functions instead of resulting indicators. For more information, see "Built-In Functions (Alphabetically)" on page 405.

A resulting indicator is defined by an entry in positions 71 through 76 of the calculation specifications. The purpose of the resulting indicators depends on the operation code specified in positions 26 through 35. See the individual operation code in Chapter 26, "Operation Code Details" for a description of the purpose of the resulting indicators. For example, resulting indicators can be used to test the result field after an arithmetic operation, to identify a record-not-found condition, to indicate an exception/error condition for a file operation, or to indicate an end-of-file condition.

The resulting indicators are 01-99 and LR.

Resulting indicators can be specified in three places (positions 71-72, 73-74, and 75-76) of the calculation specifications. The positions in which the resulting indicator is defined determine the condition to be tested.

In most cases, when a calculation is processed, the resulting indicators are set off, and, if the condition specified by a resulting indicator is satisfied, that indicator is set on. However, there some exceptions to this rule, such as "LOOKUP (Look Up a Table or Array Element)" on page 599, "SETOFF (Set Indicator Off)" on page 684, and "SETON (Set Indicator On)" on page 684. A resulting indicator can be used as

a conditioning indicator on the same calculation line or in other calculations or output operations. When it is used on the same line, the prior setting of the indicator determines whether or not the calculation is processed. If it is processed, the result field is tested and the current setting of the indicator is determined (see Figure 3).

Rules for Assigning Resulting Indicators

The following rules apply when assigning resulting indicators:

- Resulting indicators cannot be used when the result field refers to an entire array.
- If the same indicator is used to test the result of more than one operation, the last operation processed determines the setting of the indicator.
- The same indicator can be used to test for more than one condition depending on the operation specified.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C*  Two resulting indicators are used to test for the different
C*  conditions in a subtraction operation.  These indicators are
C*  used to condition the calculations that must be processed for
C*  a payroll job.  Indicator 10 is set on if the hours worked (HRSWKD)
C*  are greater than 40 and is then used to condition all operations
C*  necessary to find overtime pay.  If Indicator 20 is not on
C*  (the employee worked 40 or more hours), regular pay based on a
C*  40-hour week is calculated.
C*
C     HRSWKD        SUB       40            OVERTM            3 01020
C*
C  N20PAYRAT        MULT (H)  40            PAY               6 2
C   10OVERTM        MULT (H)  OVRRAT        OVRPAY            6 2
C   10OVRPAY        ADD       PAY           PAY
C*
C*  If indicator 20 is on (employee worked less than 40 hours), pay
C*  based on less than a 40-hour week is calculated.
C   20PAYRAT        MULT (H)  HRSWKD        PAY
C*
```

Figure 3. Resulting Indicators Used to Condition Operations

Last Record Indicator (LR)

The LR indicator can be used to end the program. This indicator is tested at the end of each action subroutine to determine if the program should be ended. For more information see “ENDACT (End of Action Subroutine)” on page 568.

Using Indicators

Indicators defined as record identifying indicators, field indicators, resulting indicators, *IN, *IN(xx), or *INxx, can be used to condition files, calculation operations, or output operations. An indicator must be defined before it can be used as a conditioning indicator. The status (on or off) of an indicator is not affected when it is used as a conditioning indicator. The status can be changed only by defining the indicator to represent a certain condition.

Field Record Relation Indicators

Field record relation indicators are specified in positions 67 and 68 of the input specifications. The valid field record relation indicators are 01-99.

Note: Field record relation indicators cannot be specified for externally described files.

Field record relation indicators associate fields with a particular record type when that record type is one of several in an OR relationship. The field described on the specification line is available for input only if the indicator specified in the field record relation entry is on or if the entry is blank. If the entry is blank, the field is common to all record types defined by the OR relationship.

Assigning Field Record Relation Indicators

Specify a record identifying indicator in positions 67 and 68 to relate a field to a particular record type. When several record types are specified in an OR relationship, all fields that do not have a field record relation indicator in positions 67 and 68 are associated with all record types in the OR relationship. To relate a field to just one record type, you enter the record identifying indicator assigned to that record type in positions 67 and 68 (see Figure 4).

An indicator (01 through 99) that is not a record identifying indicator can also be used in positions 67 and 68 to condition movement of the field from the input area to the input fields.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC.............................
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
IREPORT    AA  14    1 C5
I        OR    16    1 C6
I                                  20   30  FLDB
I                                   2   10  FLDA              07
I*
I*  Indicator 07 was specified elsewhere in the program.
I*
I                                  40   50  FLDC              14
I                                  60   70  FLDD              16
```

Figure 4. Field Record Relation

The file in Figure 4 contains two different types of records, one identified by a 5 in position 1 and the other by a 6 in position 1. The FLDC field is related by record identifying indicator 14 to the record type identified by a 5 in position 1. The FLDD field is related to the record type having a 6 in position 1 by record identifying indicator 16. This means that FLDC is found on only one type of record (that identified by a 5 in position 1) and FLDD is found only on the other type. FLDA is conditioned by indicator 07, which was previously defined elsewhere in the program. FLDB is found on both record types because it is not related to any one type by a record identifying indicator.

Indicators Conditioning Calculations

Calculation specifications in the traditional format (C specifications) can include conditioning indicators positions 9 through 11. Conditioning indicators are not used by free-form calculation specifications.

Indicators that specify the conditions under which a calculation is performed are defined elsewhere in the program.

Positions 7 and 8

Specify blanks, SR, AN or OR in positions 7 and 8 of the calculation specifications. If positions 7 and 8 are blank, the calculation is processed when specified by the program logic, by a statement in a subroutine, or by a declarative operation.

Positions 9-11

To specify indicators that control the conditions under which an operation is processed, specify positions 9 through 11 on the calculation specifications. If N is specified in position 9, the indicator should be tested for the value of off ('0'). 01-99 or LR can be specified for positions 10 through 11.

Any indicator used in positions 9 through 11 must be previously defined as one of the following types of indicators:

- Record identifying indicators (input specifications, positions 21 and 22)
- Field indicators (input specifications, positions 69 through 74)
- Resulting indicators (calculation specifications, positions 71 through 76)
- *IN array, *IN(xx) array element, or *INxx field. See "Indicators Referred to as Data" on page 26 for a description of how an indicator is defined when used with one of these reserved words.

If the indicator must be off to condition the operation, place an N in position 9. The indicators in grouped AND/OR lines must all be exactly as specified before the operation is done.

Figure 5 and Figure 6 show examples of conditioning indicators.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++Sq..RiPos1NCCPos2NCCPos3NCC.PFromTo++DField+L1M1FrPlMnZr...*
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
I*
I*  Field indicators can be used to condition operations.  Assume the
I*  program is to find weekly earnings including overtime.  The over-
I*  time field is checked to determine if overtime was entered.
I*  If the employee has worked overtime, the field is positive and -
I*  indicator 10 is set on.  In all cases the weekly regular wage
I*  is calculated.  However, overtime pay is added only if
I*  indicator 10 is on.
I*
ITIME      AB  01
I                                    1    7  EMPLNO
I                                    8   10 0OVERTM             10
I                                   15   20 2RATE
I                                   21   25 2RATEOT
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C*
C*  Field indicator 10 was assigned on the input specifications.
C*  It is used here to condition calculation operations.
C*
C                   EVAL (H)  PAY = RATE * 40
C   10              EVAL (H)  PAY = PAY + (OVERTM * RATEOT)
```

Figure 5. Conditioning Operations (Field Indicators)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC.............................
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
I*
I*  A record identifying indicator is used to condition an operation.
I*  When a record is read with a T in position 1, the 01 indicator is
I*  set on.  If this indicator is on, the field named SAVE is added
I*  to SUM.  When a record without T in position 1 is read, the 02
I*  indicator is set on.  The subtract operation, conditioned by 02,
I*  then performed instead of the add operation.
I*
IFILE      AA  01    1 CT
I         OR   02    1NCT
I                                10   15 2SAVE
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C*  Record identifying indicators 01 and 02 are assigned on the input
C*  specifications.  They are used here to condition calculation
C*  operations.
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C   01              ADD       SAVE          SUM               8 2
C   02              SUB       SAVE          SUM               8 2
```

Figure 6. Conditioning Operations (Record Identifying Indicators)

Indicators Used in Expressions

Indicators can be used as booleans in expressions in the extended-factor 2 field of the calculation specification. They must be referred to as data (that is, using *IN or *INxx). Figure 7 demonstrate this.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C* In these examples, the IF structure is performed only if 01 is on.
C* *IN01 is treated as a boolean with a value of on or off.
C* In the first example, the value of the indicator ('0' or '1') is
C* checked.
C                   IF        *IN01
C* In the second example, the logical expression B < A is evaluated.
C* If true, 01 is set on.  If false 01 is set off.  This is analogous
C* to using COMP with A and B and placing 01 in the appropriate
C* resulting indicator position.
C                   EVAL      *IN01 = B < A
```

Figure 7. Indicators Used in Expressions

See Chapter 24, “Expressions,” on page 381 and “EVAL (Evaluate Expression)” on page 571 for more information.

Indicators Conditioning Output

Indicators used to specify the conditions under which an output record or an output field is written must be previously defined in the program. Indicators to condition output are specified in positions 21 through 29. All indicators are valid for conditioning output.

The indicators you use to condition output must be previously defined as one of the following types of indicators:

- Record identifying indicators (input specifications, positions 21 and 22)
- Indicators set by the VisualAge RPG program such as 01-99

- *IN array, *IN(xx) array element, or *INxx field.

If an indicator conditions an entire record, enter the indicator on the line that specifies the record type. If an indicator conditions when a field is to be written, enter the indicator on the same line as the field name.

Conditioning indicators are not required on output lines. If conditioning indicators are not specified, the line is output every time that type of record is checked for output. If conditioning indicators are specified, one indicator can be entered in each of the three separate output indicator fields (positions 22 and 23, 25 and 26, and 28 and 29). If these indicators are on, the output operation is done. An N in the position preceding each indicator (positions 21, 24, or 27) means that the output operation is done only if the indicator is not on (a negative indicator). No output line should be conditioned by all negative indicators; at least one of the indicators should be positive.

Output indicators can be specified in an AND/OR relationship by specifying AND/OR in positions 16 through 18. An unlimited number of AND/OR lines can be used. AND/OR lines can be used to condition output records, but they cannot be used to condition fields. However, a field can be conditioned with more than three indicators by using the EVAL operation in calculations. Figure 8 illustrates this.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C* Indicator 20 is set on only if indicators 10, 12, 14,16, and 18
C* are set on.
C                   EVAL      *IN20 = *IN10 AND *IN12 AND *IN14
C                              AND *IN16 AND *IN18
C                   EXCPT
OFilename++EAddN01N02N03Excnam++++.....................................
O.............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
O* OUTFIELD is conditioned by indicator 20, which effectively
O* means it is conditioned by all the indicators in the EVAL
O* operation.
OPRINTER   E
O               20      OUTFIELD
```

Figure 8. Using EVAL with indicators

Indicators Referred to as Data

Another way of referring to and manipulating indicators is to use the *IN and *INxx reserved words.

*IN

The array *IN is a predefined array of 99 one-position, character elements representing the indicators 01 through 99. The elements of the array should contain only the character values '0' (zero) or '1' (one).

The specification of the *IN array or the *IN(xx) variable-index array element as a field in an input record, as a result field, or as factor 1 in a PARM operation defines indicators 01 through 99 for use in the program.

The operations or references valid for an array of single character elements are valid with the array *IN except that the array *IN cannot be specified as a subfield in a data structure, or as a result field of a PARM operation.

*INxx

The field *INxx is a predefined one-position character field where xx represents any one of the indicators.

The specification of the *INxx field or the *IN(n) fixed-index array element (where n = 1 - 99) as a field in an input record, as a result field, or as factor 1 in a PARM operation defines the corresponding indicator for use in the program.

Specify *INxx wherever a one-position character field is valid. *INxx cannot be specified as a subfield in a data structure, as the result field of a PARM operation, or in a SORTA operation.

Rules for Specifying Indicators Referred to as Data

The following rules apply to *IN, the array element *IN(xx) or the field *INxx:

- Moving a character '0' (zero) or *OFF to any of these fields sets the corresponding indicator off.
- Moving a character '1' (one) or *ON to any of these fields sets the corresponding indicator on.
- Do not move any value, other than '0' (zero) or '1' (one), to *INxx.
- If you take the address of *IN, *IN01 - *IN99, or *IN(index), indicators *IN01 to *IN99 will be defined. If you take the address of any other indicator, such as *INLR, only that indicator will be defined.

See Figure 9 for some examples of indicators referred to as data.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C*  When this program is called, a single parameter is passed to
C*  control some logic in the program.  The parameter sets the value
C*  of indicator 50.  The parameter must be passed with a character
C*  value of 1 or 0.
C*
C     *ENTRY        PLIST
C     *IN50         PARM                    SWITCH            1
C*
C*
C*  Subroutine SUB1 uses indicators 61 through 68.  Before the
C*  subroutine is processed, the status of these indicators used in
C*  the mainline program is saved.  (Assume that the indicators are
C*  set off in the beginning of the subroutine.)  After the subroutine
C*  is processed, the indicators are returned to their original state.
C*
C*
C                   MOVEA     *IN(61)        SAV8              8
C                   EXSR      SUB1
C                   MOVEA     SAV8          *IN(61)
C*
C*  A code field (CODE) contains a numeric value of 1 to 5 and is
C*  used to set indicators 71 through 75.  The five indicators are set
C*  off.  Field X is calculated as 70 plus the CODE field.  Field X is
C*  then used as the index into the array *IN.  Different subroutines
C*  are then used based on the status of indicators 71 through 75.
C*
C                   MOVEA     '00000'       *IN(71)
C   70              ADD       CODE          X                 3 0
C                   MOVE      *ON           *IN(X)
C   71              EXSR      CODE1
C   72              EXSR      CODE2
C   73              EXSR      CODE3
C   74              EXSR      CODE4
C   75              EXSR      CODE5
```

Figure 9. Examples of Indicators Referred to as Data

Summary of Indicators

Table 2. When Indicators Are Set On and Off

Type of Indicator	Set On	Set Off
Record identifying	Immediately after record is read	By the programmer
Field indicator	By blank or zero in specified fields, by plus in specified field, or by minus in specified field.	Before this field status is to be tested the next time.
Resulting	When the calculation is processed and the condition that the indicator represents is met.	The next time a calculation is processed for which the same indicator is specified as a resulting indicator and the specified condition is not met.
LR	By the programmer.	By the programmer.

Chapter 4. Working with Components

One of three possible target objects can result from a compilation. The result depends on the control specification keyword used:

- A component is created when the NOMAIN and EXE keywords are not present.
- A utility, or NOMAIN, DLL is created when the NOMAIN keyword is specified. This DLL contains only RPG subprocedures.
- An RPG EXE is created when the EXE keyword is specified. This module contains a main procedure and subprocedures.

This section describes how to start and stop components, as well as how to initialize and terminate components. For an overview of creating and using NOMAIN DLLs and EXEs, see Chapter 6, Chapter 6, "Subprocedures and Prototypes," on page 63.

Starting and Stopping Components

The START and STOP operation codes allow you to execute multiple components in an application. The START operation starts a new component in an application. The STOP operation terminates the execution of a component.

For more information on these operation codes, see "START (Start Component or Call Local Program)" on page 689 and "STOP (Stop Component)" on page 691.

Initializing Components

A VisualAge RPG application can consist of one or more components. Each component is started independently. The first (primary) component is started when the application is run. All subsequent (secondary) components get started by the user or the program depending on the events that occur and the action subroutines that handle the events. Secondary components can be started in any order.

The EXE file for the application invokes the primary component. Parameters can be passed to this component from the command line. Each parameter is converted from the character string entered to the target data type of the parameters on the *ENTRY PLIST.

Secondary components are invoked using the START operation code from either the primary component or from other secondary components. Parameters can be passed to secondary components using the PARM and PLIST operation codes. Parameters are not converted for a secondary component.

After a component receives any parameters, the following occurs:

1. The program fields are initialized.
2. Files are opened and data structures, prerun-time arrays and tables are loaded.
3. For any *ENTRY PLIST parameters, the result field is moved to factor 1.
4. If a user initialization subroutine (*INZSR) is specified, it is run. Most operation dealing with the component's parts and events will not work at this time because the component's run-time environment has not been initialized.
5. For any *ENTRY PLIST parameters, factor 2 is copied to the result field.
6. Data structures and variables to be used by the RESET operation are saved.
7. The component's run-time environment is initialized.

8. If action subroutines have been written for them, an initial set of events is handled for the initial set of windows and their parts. For example, any window and its parts which have startup attributes specifying "Open Immediately" cause a CREATE event. Any events generated during the execution of these action subroutines also invoke any action subroutines written for them at this time.

Once initialization for a component is complete, the component's parts are available to the application. The end user can generate events to invoke action subroutine in any of the currently opened components.

There are cases where certain operation codes, attributes and the default exception handler are not allowed during initialization of the application. For example, you cannot obtain an attribute of a part before the part has been created. For more information, see Table 3 on page 35, Table 4 on page 36, Table 5 on page 37, and Table 6 on page 39.

Terminating Components

Components are terminated by either ending the primary component or by ending a component which started one or more components. When multiple components are terminated, the components are terminated in reverse hierarchical order. Each component has its *TERMSR called (normal termination) in reverse hierarchical order. Each component, in turn, goes though its cleanup and termination (for example, closing files).

When a component ends abnormally in a multiple component application, only that component ends abnormally. Any other components that also get ended, end normally.

Normal Termination

A component terminates normally in the following situations:

- If LR is on when ENDACT is reached for the root action subroutine or if RETURN is executed from the root action subroutine.

  The root action subroutine is the subroutine at the bottom (or first) of any nested action subroutines. Nested action subroutines occur when an event invokes a new action subroutine while executing another action subroutine.

  For example, the action subroutine BUTTON+CLICK+WINDOW1 contains the SHOWWIN 'window2' operation. This causes a CREATE event which invokes the action subroutine WINDOW2+CREATE+WINDOW2. If another event occurs while the CREATE event is being handled, (for example, 'WINDOWX' SETATR 1 'FOCUS'), then the action subroutine WINDOW2+CREATE+WINDOW2 is suspended and action subroutine WINDOWX+FOCUS+WINDOWX is invoked. The call stack includes the following nested action subroutines:

  ```
  1.  FIELD1+FOCUS+WINDOWX
  2.  WINDOW2+CREATE+WINDOW2
  3.  BUTTON+CLICK+WINDOW1    (root action subroutine)
  ```

  LR is not checked until:
  1. The action subroutine WINDOWX+FOCUS+WINDOWX ends
  2. The action subroutine WINDOW2+CREATE+WINDOW2 ends
  3. The ENDACT or RETURN operation is performed for the BUTTON+CLICK+WINDOW1 action subroutine.
- If STOP is performed on the component. For more information, see "STOP (Stop Component)" on page 691.
- If the *PSSR is executed and it ends with one of the following:

- ENDSR '*DEFAULT' or an equivalent field name. The LR indicator is on at the end of the root action subroutine.
- ENDSR '*NODEFAULT' or an equivalent field name. The LR indicator is on at the end of the root action subroutine.

For more information, see "ENDSR (End of User Subroutine)" on page 569 and "Component Errors/Exceptions" on page 58.

- If the default exception handler puts up the message information window and one of the following choices is made:
  - Do Default Processing and the LR indicator is on at the end of the root action subroutine
  - Do Not Do Default Processing and the LR indicator is on at the end of the root action subroutine

For more information, see "Component Errors/Exceptions" on page 58.

The following occurs for normal termination:
- If *TERMSR exists, it is run
- Files, prerun-time arrays and tables, and data area data structures are written
- All files are closed
- All data area are unlocked.

*TERMSR is a user-written subroutine where any final code execution can occur. When *TERMSR is invoked, no action subroutines are active and the current component has been marked as being in termination. This means that few graphical user interface operations are allowed. See Table 3 on page 35, Table 4 on page 36, Table 5 on page 37, and Table 6 on page 39.

See “Component Status Codes” on page 58 for a list of status values for normal component termination.

Abnormal Termination

A component terminates abnormally if any of the following situations occurs:
- The *PSSR is executed and it ends with one of the following:
  - ENDSR ’*ENDCOMP’, ENDSR ’*CANCL’, or an equivalent field name
  - ENDSR ’*ENDAPPL’ or an equivalent field name
- The default exception handler puts up the message information window and one of the following choices is made:
  - Terminate Component
  - Terminate Application
- An abnormal condition occurs in the GUI during run time.

The following occurs for abnormal termination:
- All files are closed
- All data areas are unlocked.

Note: *TERMSR is not called for abnormal termination.

Initializing, Terminating, and Event Handling Restrictions

There are cases where certain operation codes, attributes, and the default exception handler are not allowed during some stage of an application. The following tables describes various restrictions during initialization, termination, or normal event handling.

Table 3. Operation Code Restrictions during Initialization, Termination, and Event Handling

GUI Operation	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling
CLSWIN	Not allowed	Not allowed	No restrictions
DSPLY	No restrictions	Not allowed	The information window that is displayed interferes with any events that have been posted. If the DSPLY operation is performed from the same subroutine or from a nested action subroutine after a CLSWIN or STOP operation has been performed (for example, the Close Window or Close Component events are still pending), the pending events are received by the DSPLY operation but are not performed.

SHOWWIN	Not allowed	Not allowed	The same operation code cannot be performed multiple times from within the same action subroutine or nested action subroutine. For example, an action subroutine contains:
			`SHOWWIN 'WIN1'`
			`CLSWIN 'WIN1'`
			`SHOWWIN 'WIN1'`
			In this case, the second SHOWWIN fails.

Table 3. Operation Code Restrictions during Initialization, Termination, and Event Handling (continued)

GUI Operation	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling

START	No restrictions	No restrictions	The same operation code cannot be performed multiple times from within the same action subroutine or nested action subroutine. For example, an action subroutine contains:
			`START 'COMP2'`
			`STOP 'COMP2'`
			`STOP 'COMP2'`
			In this case, the second STOP fails.

STOP 'self'	Not allowed	Not allowed	A component cannot be ended from a nested action subroutine.
STOP 'other'	Cannot end your parent component	Cannot end your parent component	A component cannot be ended from a nested action subroutine.

Table 4. Attribute Restrictions during Initialization, Termination, and Event Handling

Attribute	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling
Part attributes (GETATR, SETATR, %GETATR, %SETATR)	Not allowed	Not allowed	No restrictions
Event attributes (%PART, ...)	Not allowed	Not allowed	No restrictions
System attributes (%DSPWIDTH, %DSPHEIGHT)	Not allowed	Not allowed	No restrictions

Table 5. Default Exception Handler Restrictions during Initialization, Termination, and Event Handling

Attribute	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling
Message information window, Do default processing	No restrictions	The component is terminated and an asynchronous information window is displayed.	The information window that is displayed interferes with any events that have been posted. If this operation is performed from the same subroutine or a nested action subroutine after a CLSWIN or STOP operation has been performed (for example, the Close Window or Close Component events are still pending), the pending events are received by this operation and are not performed.
Message information window, Do not do default processing	No restrictions	The component is terminated and an asynchronous information window is displayed.	The information window that is displayed interferes with any events that have been posted. If this operation is performed from the same subroutine or a nested action subroutine after a CLSWIN or STOP operation has been performed (for example, the Close Window or Close Component events are still pending), the pending events are received by this operation and are not performed.

Table 5. Default Exception Handler Restrictions during Initialization, Termination, and Event Handling (continued)

Attribute	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling
Message information window, Terminate component	No restrictions	The component is terminated and an asynchronous information window is displayed.	A component cannot be ended from a nested action subroutine. The information window that is displayed interferes with any events that have been posted. If this operation is performed from the same subroutine or a nested action subroutine after a CLSWIN or STOP operation has been performed (for example, the Close Window or Close Component events are still pending), the pending events are received by this operation and are not performed.
Message information window, Terminate application	No restrictions	The component is terminated and an asynchronous information window is displayed.	The information window that is displayed interferes with any events that have been posted. If this operation is performed from the same subroutine or a nested action subroutine after a CLSWIN or STOP operation has been performed (for example, the Close Window or Close Component events are still pending), the pending events are received by this operation and are not performed.

Table 6. Restrictions for Ending Components during Initialization, Termination, and Event Handling

Ending a Component	Initialization (*INZSR)	Termination (*TERMSR)	Event Handling
*PSSR BEGSR.. ENDSR '*DEFAULT'	No restrictions	No restrictions	No restrictions
*PSSR BEGSR.. ENDSR '*NODEFAULT'	No restrictions	No restrictions	No restrictions
*PSSR BEGSR.. ENDSR '*ENDCOMP' or ENDSR '*CANCL'	No restrictions	No restrictions	A component cannot be ended from a nested action subroutine.
*PSSR BEGSR.. ENDSR '*ENDAPPL'	No restrictions	No restrictions	A component cannot be ended from a nested action subroutine.

Chapter 5. Error and Exception Handling

Exception/errors fall into two classes: program and file. Information on file and program exception/errors is made available to a VARPG program using file information data structures and program status data structures, respectively. File and Program exception/error subroutines may be specified to handle these types of exception/errors. This section describes error and exception handling for files, programs, and components.

File Exception/Errors

Some examples of file exception/errors are: undefined record type, an error in trigger program, an I/O operation to a closed file, a device error, and an array/table load sequence error. They can be handled in one of the following ways:

- The operation code extender 'E' can be specified. When specified, before the operation begins, this extender sets the %ERROR and %STATUS built-in functions to return zero. If an exception/error occurs during the operation, then after the operation %ERROR returns '1' and %STATUS returns the file status. The optional file information data structure is updated with the exception/error information. You can determine the action to be taken by testing %ERROR and %STATUS.
- An indicator can be specified in positions 73 and 74 of the calculation specifications for an operation code. This indicator is set on if an exception/error occurs during the processing of the specified operation. The optional file information data structure is updated with the exception/error information. You can determine the action to be taken by testing the indicator.
- ON-ERROR groups can be used to handle errors for statements processed within a MONITOR block. If an error occurs when a statement is processed, control passes to the appropriate ON-ERROR group.
- A file exception/error subroutine can be specified. The subroutine is defined by the INFSR keyword on a file description specification with the name of the subroutine that is to receive the control. Information regarding the file exception/error is made available through a file information data structure that is specified with the INFDS keyword on the file description specification. You can also use the %STATUS built-in function, which returns the most recent value set for the program or file status. If a file is specified, %STATUS returns the value contained in the INFDS *STATUS field for the specified file.
- If the indicator, 'E' extender, MONITOR block,or the file exception/error subroutine is not present, any file exception/errors are handled by the VisualAge RPG default error handler.

File Information Data Structure

The file information data structure provides information for file errors. A file information data structure (INFDS) can be defined for each file to make file exception, error, and file feedback information available to the program. This data structure must be unique for each file. It contains the following feedback information:

- File Feedback (positions 1 to 80)
- Open Feedback (positions 81 to 240)
- Input/Output Feedback (241 to 366)
- Device-Specific Feedback (position 367)

Note: The length of the INFDS depends on what fields you have declared in your INFDS.

File Feedback Information

The file feedback information starts in position 1 and ends in position 80 in the INFDS. It contains data about the file which is specific to the VisualAge RPG program, including:

- The name of the file for which the exception or error occurred
- The record being processed when the exception or error occurred or the record that caused the exception or error
- The last operation being processed when the exception or error occurred
- The status code
- The routine where the exception or error occurred.

Note: Overwriting the file feedback section can cause unexpected results in subsequent error handling and is not recommended.

The location of some of the more commonly used subfields in the file feedback section is defined by special keywords. Table 7 summarizes these keywords.

Table 7. File Feedback Information in the INFDS

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
1	8	Character	8	*FILE	The first 8 characters of the file name
9	9	Character	1		Open indication (1 = open)
10	10	Character	1		End of file (1 = end of file)
11	15	Zoned decimal	5,0	*STATUS	Status code. See "File Status Codes" on page 49.

Table 7. File Feedback Information in the INFDS (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
16	21	Character	6	*OPCODE	Operation code. The first five positions (left-adjusted) specify the type of operation by using the character representation of the calculation operation codes. For example, if a READE was being processed, READE is placed in the leftmost five positions. Operation codes which have 6 letter names are be shortened to 5 letters. **DELETE** DELET **EXCEPT** EXCPT **READPE** REDPE **UNLOCK** UNLCK **UPDATE** UPDAT The remaining position contains one of the following: **F** The last operation was specified for a file name. **R** The last operation was specified for a record. **I** The last operation was an implicit file operation.
22	29	Character	8	*ROUTINE	First 8 characters of the procedure name or zero if the call is by procedure pointer
30	37	Character	8		Source listing line number
38	42	Zoned decimal	5,0		User-specified reason for error on SPECIAL file
38	45	Character	8	*RECORD	For a program described file the record identifying indicator is placed left-adjusted in the field; the remaining six positions are filled with blanks. For an externally described file, the first 8 characters of the name of the record being processed when the exception or error occurred.

Table 7. File Feedback Information in the INFDS (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
46	52	Character	7		Machine or system message number
53	66	Character	14		Unused

For a complete description of the contents of the file feedback area, see the *DB2® Universal Database™* section of the *Database and File Systems* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**.

INFDS File Feedback Example: To define an INFDS which contains fields in the file feedback section, specify the following entries:

- Specify the INFDS keyword on the file description specification with the name of the file information data structure
- Specify the file information data structure and the subfields you wish to use on a definition specification
- Specify special keywords left-adjusted, in the FROM field (positions 26-32) on the definition specification, or specify the positions of the fields in the FROM field (position 26-32) and the TO field (position 33-39).

```
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    IF   E             DISK    INFDS(FILEFBK)    REMOTE
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++Comments++++++++++
DFILEFBK          DS
D FILE              *FILE                                          * File name
D OPEN_IND               9      9                                  * File open?
D EOF_IND               10     10                                  * File at eof?
D STATUS            *STATUS                                        * Status code
D OPCODE            *OPCODE                                        * Last Opcode
D ROUTINE           *ROUTINE                                       * RPG Routine
D LIST_NUM               30     37                                 * Listing line
D SPCL_STAT              38     42S 0                              * SPECIAL status
D RECORD            *RECORD                                        * Record name
D MSGID                  46     52                                 * Error MSGID
```

Figure 10. Example of Coding an INFDS with File Feedback Information

Note: The keywords are not labels and cannot be used to access the subfields. Short entries are padded on the right with blanks.

Open Feedback Information

Positions 81 through 240 in the file information data structure contain open feedback information. The contents of this area are copied to the open feedback section whenever the file associated with the INFDS is opened. This includes members opened as a result of a read operation on a multi-member processed file.

Note: Open feedback information is not provided for printer files, however device feedback information is provided for printer files. See the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter** for a complete description of the contents of the open feedback area.

INFDS Open Feedback Example: To define an INFDS which contains fields in the open feedback section, specify the following entries:

- Specify the INFDS keyword on the file description specification with the name of the file information data structure
- Specify the file information data structure and the subfields you wish to use on a definition specification.
- Use information in the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** to determine which fields you wish to include in the INFDS. To calculate the From and To positions (positions 26 through 32 and 33 through 39 of the definition specifications) that specify the subfields of the open feedback section, use the Offset, Data Type, and Length given in the **Information Center** and do the following calculations:

```
From = 81 + Offset
To = From - 1 + Character_Length
Character_Length = Length (in bytes)
```

Input/Output Feedback Information

Positions 241 through 366 in the file information data structure are used for input/output feedback information. The contents of the file common input/output feedback area are copied to the input/output feedback section only after a POST for the file. For more information see "POST (Post)" on page 652.

A description of the contents of the input/output feedback area can be found in the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center**.

Note: I/O feedback information is not provided for printer files, however device-specific feedback information is provided for printer files.

INFDS Input/Output Feedback Example: To define an INFDS which contains fields in the open feedback section, specify the following entries:

- Specify the INFDS keyword on the file description specification with the name of the file information data structure
- Specify the file information data structure and the subfields you wish to use on a definition specification.
- Use information in the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** to determine which fields you wish to include in the INFDS. To calculate the From and To positions (positions 26 through 32 and 33 through 39 of the definition specifications) that specify the subfields of the input/output feedback section, use the Offset, Data Type, and Length given in **Information Center** and do the following calculations:

```
From = 241 + Offset
To = From - 1 + Character_Length
Character_Length = Length (in bytes)
```

  For example, for device class of a file, **Information Center** gives:

```
   Offset = 30
   Data Type is character
   Length = 2
Therefore,
   From = 241 + 30 = 271,
   To = 271 - 1 + 2 = 272.
```

See subfield DEV_CLASS in Figure 11 on page 46.

```
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++Comments++++++++++++
FMYFILE    IF   E             DISK    INFDS(MYIOFBK)    REMOTE
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++Comments++++++++++
DMYIOFBK          DS
D                                                                          * 241-242 not used
D WRITE_CNT             243    246B 0                                      * Write count
D READ_CNT              247    250B 0                                      * Read count
D WRTRD_CNT             251    254B 0                                      * Write/read count
D OTHER_CNT             255    258B 0                                      * Other I/O count
D OPERATION             260    260                                         * Current operation
D IO_RCD_FMT            261    270                                         * Rcd format name
D DEV_CLASS             271    272                                         * Device class
D IO_PGM_DEV            273    282                                         * Pgm device name
D IO_RCD_LEN            283    286B 0                                      * Rcd len of I/O
```

Figure 11. Coding Input/Output Feedback Information

Device-Specific Feedback Information

The device-specific feedback information in the file information data structure starts at position 367 in the INFDS. It contains input/output feedback information specific to a database or printer device.

The length of the INFDS when device-specific feedback information is required, is variable and depends on whether the device type of the file is variable and on whether the file is keyed or not (if it's a DISK file).

For externally-described DISK files, the INFDS is at least long enough to hold the longest key in the file beginning at position 401.

The contents of the device-specific input/output feedback area of the file are copied to the device-specific feedback section of the INFDS only after a POST for the file. For more information, see "POST (Post)" on page 652.

INFDS Device-Specific Feedback Examples: To define an INFDS which contains fields in the device feedback section, specify the following entries:

- Specify the INFDS keyword on the file description specification with the name of the file information data structure
- Specify the file information data structure and the subfields you wish to use on a definition specification
- Use information in the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** to determine which fields you wish to include in the INFDS. To calculate the From and To positions (positions 26 through 32 and 33 through 39 of the definition specifications) that specify the subfields of the device-specific feedback, use the Offset, Data Type, and Length given in the **Information Center** and do the following calculations:

  ```
  From = 367 + Offset
  To = From - 1 + Character_Length
  Character_Length = Length (in bytes)
  ```

  For example, for relative record number of a data base file, the **Information Center** uses:

  ```
  Offset = 30
  Data Type is binary
  Length = 4
Therefore,
  From = 367 + 30 = 397,
  To = 397 - 1 + 4 = 400.
  ```

See subfield DB_RRN in the DBFBK data structure in Figure 12.

```
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    IF   E             DISK    INFDS(DBFBK)    REMOTE
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++Comments++++++++++
DDBFBK            DS
D FDBK_SIZE             367    370B 0                                    * Size of DB fdbk
D JOIN_BITS             371    374B 0                                    * JFILE bits
D LOCK_RCDS             377    378B 0                                    * Nbr locked rcds
D POS_BITS              385    385                                       * File pos bits
D DLT_BITS              384    384                                       * Rcd deleted bits
D NUM_KEYS              387    388B 0                                    * Num keys (bin)
D KEY_LEN               393    394B 0                                    * Key length
D MBR_NUM               395    395B 0                                    * Member number
D DB_RRN                397    400B 0                                    * Relative-rcd-num
D KEY                   401   2400                                       * Key value (max
D                                                                       *   size 2000)
```

Figure 12. Example of Coding an INFDS with Database Specific Feedback Information

Blocking Considerations: The fields of the input/output specific feedback area and in most cases the fields of the device-specific feedback information area, are not updated for each operation to the file in which the records are blocked and unblocked, except for key and relative record number. The exception to this occurs when a POST operation is performed. In this case, all of the fields of the input/output specific and device-specific feedback areas are updated. In a POST operation, the key and relative record number are updated with information from the current record, not the last record in the block.

File Exception and Error Subroutine (INFSR)

To identify the subroutine that receives control following any file exceptions or errors, specify the INFSR keyword on the File Description specification with the name of the subroutine. The subroutine name can be *PSSR, which indicates that the program exception/error subroutine is given control for the exception and errors on this file.

A file exception/error subroutine receives control when an exception or error occurs on a file operation that does not have an indicator specified in positions 73 and 74, does not have an (E) extender, and is not in the monitor block of a MONITOR group that can handle the error.. The file exception/error subroutine can also be run by the EXSR operation code. Any of the operation codes can be used in the file exception/error subroutine. Factor 1 of the BEGSR operation and factor 2 of the EXSR operation must contain the name of the subroutine that receives control (same name as specified with the INFSR keyword on the file description specifications). The ENDSR operation must be the last specification for the file exception/error subroutine and must be specified as follows:

Position	Entry
6	C
7-11	Blank
12-25	Can contain a label that is used in a GOTO specification within the subroutine.
26-35	ENDSR
36-49	Optional entry to designate where control is to be returned

following processing of the subroutine. The entry must be a character field, literal, or array element whose value specifies one of the following return points.

Note: If the return points are specified as literals, they must be enclosed in apostrophes. If they are specified as named constants, the constants must be character and must contain only the return point with no leading blanks. If they are specified in fields or array elements, the value must be left-adjusted in the field or array element.

***DEFAULT**
Return control from the current action subroutine and perform the default processing associated with the current event.

***NODEFAULT**
Return control from the current action subroutine. Do not perform any default processing. If LR is on when processing reaches this point, the component is terminated, and the *DEFAULT and *NODEFAULT return points are ignored.

***CANCL**
Terminate the component abnormally.

***ENDAPPL**
Terminate all currently active components, ending the application.

***ENDCOMP**
Terminate the component abnormally.

Blanks
Return control to the default error handler. This applies when factor 2 is a value of blanks and when factor 2 is not specified. If the subroutine was called by the EXSR operation and factor 2 is blank, control returns to the next sequential instruction. Blanks are only valid at run time.

50-76 Blank.

Remember the following when specifying the file exception/error subroutine:

- You can explicitly call the file exception/error subroutine by specifying the name of the subroutine in factor 2 of the EXSR operation.
- After the ENDSR operation of the file exception/error subroutine is run, the field or array element in factor 2 is reset to blanks. If you do not place a value in this field during the processing of the subroutine, the default error handler receives control following processing of the subroutine unless the subroutine was called by the EXSR operation. Because factor 2 is set to blanks, you can specify the return point within the subroutine that is best suited for the exception or error that occurred. If the subroutine was called by the EXSR operation, control returns to the next sequential instruction following the EXSR operation. A file exception/error subroutine can handle errors in more than one file.
- If a file exception or error occurs during the start or end of a program, control passes to the default error handler, and not to the user-written file exception /error or subroutine (INFSR).

- Because the file exception/error subroutine may receive control whenever a file exception or error occurs, an exception or error could occur while the subroutine is running if an I/O operation is processed on the file in error. If an exception/error occurs on the file already in error while the subroutine is running, the subroutine is called again; this results in a program loop unless you code the subroutine to avoid this problem. One way to avoid such a program loop is to set a first-time switch in the subroutine. If it is not the first time through the subroutine, set the LR indicator on and issue the RETURN operation as follows:

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* If INFSR is already handling the error, exit.
C     ERRRTN        BEGSR
C     SW            IFEQ      '1'
C                   SETON                                        LR
C                   RETURN
C* Otherwise, flag the error handler.
C                   ELSE
C                   MOVE      '1'           SW
C                     :
C                     :
C                     :
C                   ENDIF
C* End error processing.
C                   MOVE      '0'           SW
C                   ENDSR
```

Note: It may not be possible to continue processing the file after an I/O error has occurred. To continue, it may be necessary to issue a CLOSE operation and then an OPEN operation to the file.

File Status Codes

Any code placed in the subfield location *STATUS that is greater than 99 is considered to be an exception or error. When the status code is greater than 99; the error indicator — if specified in positions 73 and 74 — is set on, or the %ERROR built-in function — if the 'E' extender is specified — is set to return '1'. Otherwise, the file exception/error subroutine receives control. Location *STATUS is updated after every file operation.

You can use the %STATUS built-in function to get information on exception/errors. It returns the most recent value set for the program or file status. If a file is specified, %STATUS returns the value contained in the INFDS *STATUS field for the specified file.

The following tables summarize the codes placed in the subfield location *STATUS for the file information data structure:

Table 8. Normal Codes

Code	Device[1]	RC	Condition
00000			No exception/error
00011	D		End of file on a read (input)
00012	D		No-record-found condition on a CHAIN, SETLL, and SETGT operations
00014			Output record of local file truncated
00015			Input record of local file truncated
Note: [1]"Device" refers to the devices for which the condition applies. The following abbreviations are used: P = PRINTER; D = DISK; SP = SPECIAL			

Table 9. Exception/Error Codes

Code	Device[1]	RC	Condition
01011	D		Undefined record type (input record does not match record identifying indicator)
01021	D		Tried to write a record that already exists (file being used has unique keys and key is duplicate)
01022	D		Referential constraint error detected on file member
01041	n/a		Array/table load sequence error
01042	n/a		Array/table load sequence error
01051	n/a		Excess entries in array/table file
01211	all		I/O operation to a closed file
01215	all		OPEN issued to a file already opened
01216[2]	all		Error on an implicit OPEN/CLOSE operation.
01217[2]	all		Error on an explicit OPEN/CLOSE operation.
01218	D		Record already locked
01221	D		Update operation attempted without a prior read
01222	D		Record cannot be allocated due to referential constraint error
01231	SP		Error on SPECIAL file
01235	P		Error in PRTCTL space or skip entries

Table 9. Exception/Error Codes (continued)

Code	Device[1]	RC	Condition
01299	D,P		Other I/O error detected. For local files, this message contains one of the following ids: • *LF0001: Could not open file • *LF0002: Could not close file • *LF0003: Unexpected I/O result • *LF0004: File pointer could not be set • *LF0005: Read failed • *LF0006: Write failed • *LF0007: Could not determine size of file • *LF0008: Could not resize file • *LF0009: Could not copy file • *LF0010: Could not delete file • *LF0011: File designated as local at compile time is found to be a remote file at run time.
Note: [1]"Device" refers to the devices for which the condition applies. The following abbreviations are used: P = PRINTER; D = DISK; SP = SPECIAL; [2]Any errors that occur during an open or close operation will result in a *STATUS value of 1216 or 1217.			

Program Exception and Errors

Some examples of program exception and errors are: division by zero, SQRT of a negative number, invalid array index, an error on a CALL, an error return from a called program, and a start position or length out of range for a string operation. They can be handled in one of the following ways:

- An indicator can be specified in positions 73 and 74 of the calculation specifications for certain operation codes. This indicator is set on if an exception or error occurs during the processing of the specified operation. The optional program status data structure is updated with the exception/error information. You can determine the action to be taken by testing the indicator.
- The operation code extender 'E' can be specified for some operation codes. When specified, before the operation begins, this extender sets the %ERROR and %STATUS built-in functions to return zero. If an exception/error occurs during the operation, then after the operation %ERROR returns '1' and %STATUS returns the program status. The optional program status data structure is updated with the exception/error information. You can determine the action to be taken by testing %ERROR and %STATUS.
- ON-ERROR groups can be used to handle errors for statements processed within a MONITOR block. If an error occurs when a statement is processed, control passes to the appropriate ON-ERROR group.
- A program exception/error subroutine can be specified by coding *PSSR in factor 1 of a BEGSR operation. Information regarding the program exception/error is made available through a program status data structure that is specified with an S in position 23 of the data structure statement on the definition specifications.
- If the indicator, 'E' extender, monitor block, or the program exception/error subroutine is not present, program exception and errors are handled by the default error handler.

Program Status Data Structure

A program status data structure can be defined to make program exception and error information available to a VisualAge RPG program.

A data structure is defined as a program status data structure by an S in position 23 of the data structure statement. A program status data structure contains subfields that provide you with information about the program exception or error that occurred. The location of these subfields is defined by special keywords or by predefined From and To positions. In order to access the subfields, you assign a name to each subfield. The keywords must be specified, left-adjusted in positions 26 through 39.

Table 10 provides the layout of the subfields of the data structure and the From and To positions of its subfields.

Table 10. Contents of the Program Status Data Structure

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
1	10	Character	10	*PROC	Component name
11	15	Zoned decimal	5,0	*STATUS	Status code
16	20	Zoned decimal	5,0		Previous status code
21	28	Character	8		Source listing line number
29	36	Character	8	*ROUTINE	Name of the routine where the exception or error occurred. This subfield is updated at the beginning of a routine or after a program call only when the *STATUS subfield is updated with a nonzero value. The following names identify the routines: ***INIT** Program initialization ***TERM** Program ending ***ROUTINE** Name of program or procedure called (first 8 characters).
37	39	Zoned decimal	3,0	*PARMS	Number of parameters passed to this program from a calling program
40	42	Character	3		Exception type: CPF for an OS/400® system exception, MCH for a machine exception or *RT for an error return code from a runtime routine. For a Windows exception, this field contains *EX.
43	46	Character	4		Exception number: For a CPF exception, this field contains a CPF message number. For a machine exception, it contains a machine exception number. For a Windows exception, this field contains the exception number in binary 9,0 format. The error return code from a VisualAge RPG runtime routine is also contained in this field, in binary 9,0 format.
47	90		44		Reserved
91	170	Character	80		Retrieved exception data. OS/400 messages are placed in this subfield
171	190		20		Reserved

Table 10. Contents of the Program Status Data Structure (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
191	198	Character	8		Date (*DATE format) the job entered the system.The date represented by this value is the same date represented by positions 270 - 275.
199	200	Zoned decimal	2,0		First 2 digits of a 4-digit year. The same as the first 2 digits of *YEAR.This field applies to the century part of the date in positions 270 to 275. For example, for the date 1999-06-27, UDATE would be 990627, and this century field would be 19. The value in this field in conjunction with the value in positions 270 - 275 has the combined information of the value in positions 191 -198. **Note:** This century field does not apply to the dates in positions 276 to 281, or positions 288 to 293.
201	208	Character	8		Name of file on which the last file operation occurred (updated only when an error occurs)
209	243	Character	35		Status information on the last file used. This information includes the status code, the operation code, the VisualAge RPG routine name, the source listing line number, and record name. It is updated only when an error occurs. **Note:** The opcode name is in the same form as *OPCODE in the INFDS.
244	253		10		Reserved
254	263	Character	10		The iSeries host Sign-On userid for a remote file open operation. This value is updated only when a different host is accessed with a different Sign-On userid.
264	269		10		Reserved
270	275	Zoned decimal	6,0		Date (in UDATE format) the program started running in the system. (UDATE is derived from this date.) See "User Date Special Words" on page 8 for a description of UDATE. This is commonly known as the 'job date'. The date represented by this value is the same date represented by positions 191 - 198.
276	281	Zoned decimal	6,0		Date of program running (the system date in UDATE format) If the year part of this value is between 40 and 99, the date is between 1940 and 1999. Otherwise the date is between 2000 and 2039. The 'century' value in positions 199 - 200 does not apply to this field.
282	287	Zoned decimal	6 (zero decimal positions)		Time of program running in the format hhmmss

Table 10. Contents of the Program Status Data Structure (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
288	293	Character	6		Date (in UDATE format) the program was compiled If the year part of this value is between 40 and 99, the date is between 1940 and 1999. Otherwise the date is between 2000 and 2039. The 'century' value in positions 199 - 200 does not apply to this field.
294	299	Character	6		Time (in the format hhmmss) the program was compiled
300	303	Character	4		Level of the compiler
304	313	Character	10		Source file name (first 10 characters)
314	429		116		Reserved

Program Status Codes

Any code placed in the subfield location *STATUS that is greater than 99 is considered to be an exception or error condition. When the status code is greater than 99; the error indicator — if specified in positions 73 and 74 — is set on, or the %ERROR built-in function — if the 'E' extender is specified — is set to return '1', or control passes to the appropriate ON-ERROR group within a MONITOR block; otherwise, the program exception/error subroutine receives control. *STATUS is updated when an exception or error occurs.

The %STATUS built-in function returns the most recent value set for the program or file status.

The following codes are placed in the subfield location *STATUS for the program status data structure:

Normal Codes:

Code	Condition
00000	No exception/error occurred
00031	Component is terminating; LR indicator on when a RETURN or ENDACT operation performed
00032	Component is terminating as a result of an explicit termination of the component (STOP component)
00033	Component is terminating as a result of an implicit termination of the component (STOP parent or grandparent of component)
00034	Component is terminating as a result of an explicit termination request from another component (STOP component)
00035	Component is terminating as a result of an implicit termination request from another component (STOP parent of component)
00050	Conversion resulted in substitution.

Exception/Error Codes:

Code	Condition
00100	Value out of range for string operation
00101	Negative square root
00102	Divide by zero
00103	An intermediate result is not large enough to contain the result
00104	Float underflow. An intermediate value is too small to be contained in the intermediate result field.
00105	Invalid characters in character to numeric conversion functions.
00112	Invalid Date, Time or Timestamp value.
00113	Date overflow or underflow. (For example, when the result of a Date calculation results in a number greater than *HIVAL or less than *LOVAL)
00114	Date mapping errors, where a Date is mapped from a 4 character year to a 2 character year and the date range is not 1940-2039
00115	Variable-length character or graphic field has a current length that is not valid.
00120	Table or array out of sequence
00121	Array index not valid
00122	OCCUR outside of range
00123	Reset attempted during initialization step of program
00202	Called program or procedure failed
00211	Error calling program or procedure
00221	Called program tried to use a parameter not passed to it
00222	Pointer or parameter error
00301	Class or method not found for a method call, or error in method call.

00302 Error while converting a Java array to an RPG parameter on entry to a Java native method.

00303 Error converting RPG parameter to Java array on exit from an RPG native method.

00304 Error converting RPG parameter to Java array in preparation for a Java method call.

00305 Error converting Java array to RPG parameter or return value after a Java method.

00306 Error converting RPG return value to Java array.

00333 Error on DSPLY operation

00401 Data area specified on IN/OUT not found

00411 Data area type or length does not match

00412 Data area not locked for output

00413 Error on IN/OUT operation

00414 User not authorized to use data area

00415 User not authorized to change data area

00421 Error on UNLOCK operation

00431 Data area previously locked by another program

00432 Data area locked by program in the same process

00451 Conversion between two CCSIDs is not supported.

00501 Failure to retrieve sort sequence

00802 Commitment control not active

00803 Rollback operation failed

00804 Error occurred on COMMIT operation

00805 Error occurred on ROLBK operation

00907 Decimal data error (digit or sign not valid)

00940 Error occurred in host services

00970 The level number of the compiler used to generate the program does not agree with the level number of the VisualAge RPG runtime subroutines.

01400 Attribute name is not valid

01401 SHOWWIN operation attempted on an opened window

01402 Part name was not found in the application

01403 New attribute value is not within the valid range

01404 Attribute access type not valid for the operation

01405 Data type of event attribute is not compatible with the operation

01406 Invalid message identifier

01407 Data type of attribute is not compatible with the operation

01408 Insufficient resources

01410 START operation failed

01411 STOP operation failed

01420 Error occurred on subfile operation

01421 The user cancelled the signon dialog

01422 The component containing the part being operated on has not been started.

1601 One or more of the DB2 product's dynamic link libraries (DLL) could not be found.

08888 Recursion error

09001 No error indicator or *PSSR

09998 Internal failure in VisualAge RPG compiler or in runtime subroutines

09999 Program exception in system routine.

Program Status Data Structure Example

To specify a program status data structure (PSDS) in your program, code the program status data structure and the subfields you wish to use on a definition specification.

```
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++Comments++++++++++
DMYPSDS          SDS
D PROC_NAME         *PROC                                                * Component name
D PGM_STATUS        *STATUS                                              * Status code
D PRV_STATUS             16     20S 0                                    * Previous status
D LINE_NUM               21     28                                       * Src list line num
D ROUTINE           *ROUTINE                                             * Routine name
D PARMS             *PARMS                                               * Num passed parms
D EXCP_TYPE              40     42                                       * Exception type
D EXCP_NUM               43     46                                       * Exception number
D*
D EXCP_DATA              91    170                                       * Exception data
D*
D DATE                  191    198                                       * Date (*DATE fmt)
D YEAR                  199    200S 0                                    * Year (*YEAR fmt)
D LAST_FILE             201    208                                       * Last file used
D FILE_INFO             209    243                                       * File error info
D*
D JOB_DATE              270    275S 0                                    * Date (UDATE fmt)
D RUN_DATE              276    281S 0                                    * Run date (UDATE)
D RUN_TIME              282    287S 0                                    * Run time (UDATE)
D CRT_DATE              288    293                                       * Create date
D CRT_TIME              294    299                                       * Create time
D CPL_LEVEL             300    303                                       * Compiler level
D SRC_FILE              304    313                                       * Source file
D*
```

Figure 13. Example of Coding a PSDS

Note: The keywords are not labels and cannot be used to access the subfields. Short entries are padded on the right with blanks.

Program Exception and Error Subroutine

To identify the subroutine that receives control when a program exception or error occurs, specify *PSSR in factor 1 of the subroutine's BEGSR operation. If an indicator is not specified in positions 73 and 74 for an operation code, or if the operation does not have an (E) extender, or if the statement is not in a MONITOR block that can handle the error, or if an exception occurs that is not expected for an operation code (for example an array indexing error during a SCAN operation),

control is transferred to this subroutine when a program exception or error occurs. In addition, the subroutine can also be called by the EXSR operation. *PSSR can be specified on the INFSR keyword on the file description specifications and receives control if a file exception/error occurs.

Any operation codes can be used in the program exception/error subroutine. The ENDSR operation must be the last specification for the subroutine, and the factor 2 entry on the ENDSR operation specifies the return point following the running of the subroutine. For more information, see "File Exception and Error Subroutine (INFSR)" on page 47.

Remember the following when specifying a program exception/error subroutine:

- You can explicitly call the *PSSR subroutine by specifying *PSSR in factor 2 of the EXSR operation.
- After the ENDSR operation of the *PSSR subroutine is run, the field, subfield, array element, or array element specified in factor 2 is reset to blanks. This allows you to specify the return point within the subroutine that is best suited for the exception or error that occurred. If factor 2 contains blanks at the end of the subroutine, the default error handler receives control; if the subroutine was called by an EXSR or CASxx operation, control returns to the next sequential instruction following the EXSR or ENDCS. If the exception occured in a subprocedure an no GOTO operation was encountered before the ENDSR operation, error code 9001 is issued and the application ends. Factor 2 is not supported on the ENDSR operation of subprocedure *PSSRs.
- Because the program exception/error subroutine may receive control whenever a non-file exception/error occurs, an exception or error could occur while the subroutine is running. If an exception/error occurs while the subroutine is running, the subroutine is called again; this results in a program loop unless you code the subroutine to avoid this problem.
- A *PSSR can be defined in a subprocedure, and each subprocedure can have its own *PSSR. Note that the *PSSR in a subprocedure is local to that subprocedure. If you want the subprocedures to share the same exception routine, then you should have each *PSSR call a shared procedure.
- If you have a *PSSR that is not defined within a subprocedure, this *PSSR is never executed in an exception occurs within a subprocedure.

Component Errors/Exceptions

The following sections describe how to handle errors during an event and which exceptions are trapped by the VisualAge RPG exception handler.

Component Status Codes

The following *STATUS values allow you to query how the component has terminated for normal termination:

00031 The component terminates because LR is on. LR is checked when the root ENDACT has been reached. The root action subroutine is the subroutine at the bottom (or first) of any nested action subroutines.

00032 The component terminates itself directly. For example, The component 'thiscomp' issues STOP 'thiscomp'. The component 'thiscomp' is terminated.

00033 The component terminates itself indirectly. For example, STOP 'myparent' is issued by the current component to terminate the component which STARTed the current component. All the children of 'myparent' are terminated first including the current component.

00034 The component is terminated directly by another component. For example, STOP 'X' is issued by another component to terminate the current component, 'X'.

00035 The component is terminated indirectly by another component. For example, STOP 'myparent' is issued by another component to terminate the parent of the current component, 'myparent', and indirectly, the current component is also being terminated.

When normal termination occurs, subroutine *TERMSR is called. *TERMSR is a user written subroutine from which any final code execution can occur. At the time that *TERMSR is invoked, no action subroutines are active, and the current component has been marked as being in termination. This means that few graphical user interface operations are allowed. See the following for more information:

- Table 3 on page 35
- Table 4 on page 36
- Table 5 on page 37
- Table 6 on page 39

Event Error Handling

If an error occurs during the handling of an event, one of two things happens:

- If a *PSSR or INFSR is not present, the default exception handler is invoked.
- If an error handling routine is present (*PSSR or INFSR), the error handling routine is invoked.

If your application contains an error handling subroutine, this subroutine continues to execute until one of the following operation codes is reached:

RETURN Control returns to the same place where ENDACT *DEFAULT processing occurs. If there are no other nested action subroutines, LR is checked:

- If LR is on, the component terminates normally. See "Normal Termination" on page 32.
- If LR is not on, the current action subroutine ends and any default action for the event is performed.

STOP The component terminates normally. Some restrictions apply. See Chapter 4, "Working with Components," on page 31.

ENDSR What you specify in factor 2 affects the flow of execution:

- If factor 2 is not specified, the default exception handler inquiry message information window is displayed.
- If *DEFAULT is specified in factor 2, control returns to the same place that ENDACT *DEFAULT processing occurs. If there are no other nested action subroutines, LR is checked:
  - If LR is on, the component terminates normally. See "Normal Termination" on page 32.
  - If LR is not on, the current action subroutine ends and any default action for the event is performed.
- If *NODEFAULT is specified in factor 2, control returns to the same place where ENDACT *NODEFAULT processing occurs. See "ENDSR (End of User Subroutine)" on page 569. If there are no other nested action subroutines, LR is checked:
  - If LR is on, the component terminates normally. See "Normal Termination" on page 32.

  - If LR is not on, the current action subroutine ends and any default action for the event is NOT performed.
- If *ENDCOMP or *CANCL is specified in factor 2, the action subroutine that was running when the error occurred finishes and the component terminates abnormally. See "ENDSR (End of User Subroutine)" on page 569.
- If *ENDAPPL is specified in factor 2, the action subroutine that was running when the error occurred is finished, and all components in the application are closed in reverse hierarchical order. See "ENDSR (End of User Subroutine)" on page 569. The component that was active when the error occurred is terminated abnormally. All other components terminate normally. See "Normal Termination" on page 32 and "Abnormal Termination" on page 34.

When the default exception handler is invoked for an exception that occurs outside a procedure, a window is displayed from which you can make one of the following choices:
- Do Default Processing (the information above for ENDSR *DEFAULT applies)
- Do Not Do Default Processing (the same information above for ENDSR *NODEFAULT applies)
- Retry the Operation: This option only appears for a small set of I/O errors. It allows you to retry the same operation.
- Terminate the Component (the same information above for ENDSR *ENDCOMP applies)
- Terminate the Application (the information above for ENDSR *ENDAPPL applies)

Note: If the exception occurs within a subprocedure and there is no local *PSSR or error indicator, the application ends.

When control is given to an error handling routine or to the default exception handler, the current action subroutine that caused the error is still active. You can still access the same event attributes that were valid at the time of the error. For example, the %BUTTON event attribute is valid during the processing of the MouseDown event. If an error occurs during the handling of this event, the %BUTTON can be referenced in the *PSSR.

Note: If %BUTTON is referenced in the *PSSR for an event where the event attribute is not valid, then an error occurs. This kind of error can easily cause the application to go into an endless recursion situation if the *PSSR is not properly coded to handle this.

For cases where multiple action subroutines are nested, the error handling routine only affects the top-most action subroutine invocation when ENDSR *DEFAULT, *NODEFAULT or an equivalent field name is executed. For example, if a SHOWWIN WINDOW2 is performed from inside the action subroutine BUTTON+CLICK+WINDOW1, then BUTTON+CLICK+WINDOW1 is suspended and the action subroutine WINDOW2+CREATE+WINDOW2, is invoked. If an error occurs during the invocation of this second action subroutine, the *PSSR or the default exception handler is 0xC0000095invoked. If *DEFAULT is taken, only WINDOW2+CREATE+WINDOW2 ends, and control returns back to BUTTON1+CLICK+WINDOW1 at the operation following SHOWWIN WINDOW2.

Exception Handling

The following exceptions are trapped by the VisualAge RPG exception handler. These exceptions are placed in the exception number field (43-46) of the PSDS as a 4 byte binary number with *EX placed in the exception type field (40-42) of the PSDS.

Access violation	0xC0000005
Integer divide by zero	0xC000009B
Float divide by zero	0xC0000095
Float invalid operation	0xC0000097
Illegal instruction	0xC000001C
Privileged instruction	0xC000009D
Integer overflow	0xC000009C
Float overflow	0xC0000098
Float underflow	0xC000009A
Float denormal operand	0xC0000094
Float inexact result	0xC0000096
Float stack check	0xC0000099
Datatype misalignment	0xC000009E
Invalid lock sequence	0xC000001D
Array bounds exceeded	0xC0000093

For more information on Windows–specific exceptions, consult the operating system's documentation.

All other exceptions are handled in one of the following ways:

- If the exception occurs during a CALLB or CALL, the status code is set to 202 or 211.
- If the exception does not occur during a CALLB or CALL, the exceptions are mapped to a status code as follows:

Integer divide by zero	102
Float divide by zero	102
Float overflow	103
Access violation	222
Datatype misalignment	222
All other exceptions	9999

Chapter 6. Subprocedures and Prototypes

One of three possible target objects can result from a compilation. The result depends on the control specification keyword used:
- A component is created when the NOMAIN and EXE keywords are not present.
- A utility, or NOMAIN, DLL is created when the NOMAIN keyword is specified. This DLL contains only RPG subprocedures.
- An RPG EXE is created when the EXE keyword is specified. This module contains a main procedure and subprocedures.

A VisualAge RPG program consists of one or more modules. A **procedure** is any piece of code that can be called with the CALLP operation code. VisualAge RPG has two kinds of procedures: a main procedure and a subprocedure. A **main procedure** is a procedure that can be specified as the program entry procedure and receives control when it is first called. Note that a main procedure is only produced when creating an EXE.

A **subprocedure** is a procedure specified after the main source section. (See "Placement of Definitions and Scope" on page 256 for the layout of the main source section for each type of compilation target.) Subprocedures differ from a main procedure in that:
- Names that are defined within a subprocedure are not accessible outside the subprocedure.
- The call interface must be prototyped.
- Calls to subprocedures must be bound procedure calls.
- Only P, D, and C specifications can be used.

All subprocedures must have a corresponding prototype in the definition specifications of the main source section. The prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters.

This section discusses the following aspects of subprocedures:
- Subprocedure definition
- NOMAIN and EXE modules
- Comparison with subroutines

Subprocedure Definition

Subprocedures are defined after the main source section. Figure 14 shows a subprocedure, highlighting the different parts of it.

```
 * Prototype for procedure FUNCTION
 *
D FUNCTION        PR            10I 0                    1
D    TERM1                       5I 0 VALUE
D    TERM2                       5I 0 VALUE
D    TERM3                       5I 0 VALUE
 *
P Function        B                                      2
 *
 *-------------------------------------------------------------
 * This procedure performs a function on the 3 numeric values
 * passed to it as value parameters.
 *
 * This illustrates how a procedure interface is specified for a
 * procedure and how values are returned from a procedure.
 *-------------------------------------------------------------
 *
D Function        PI            10I 0                    3
D    Term1                       5I 0 VALUE
D    Term2                       5I 0 VALUE
D    Term3                       5I 0 VALUE
D Result          S             10I 0                    4
C                   EVAL      Result = Term1 ** 2 * 17
C                                    + Term2      * 7    5
C                                    + Term3
C                   RETURN    Result * 45 + 23
P                 E                                      6
 *
```

Figure 14. Example of a Subprocedure

1. A Prototype which specifies the name, return value if any, and parameters if any.
2. A Begin-Procedure specification (B in position 24 of a procedure specification)
3. A Procedure-Interface definition, which specifies the return value and parameters, if any. The procedure interface must match the corresponding prototype. The procedure-interface definition is optional if the subprocedure does not return a value and does not have any parameters that are passed to it.
4. Other definition specifications of variables, constants, and prototypes needed by the subprocedure. These definitions are local definitions.
5. Any calculation specifications, standard or free-form, needed to perform the task of the procedure. The calculations may refer to both local and global definitions. Any subroutines included within the subprocedure are local. They cannot be used outside of the subprocedure. If the subprocedure returns a value, then the subprocedure must contain a RETURN operation.
6. An End-Procedure specification (E in position 24 of a procedure specification)

Except for the procedure-interface definition, which may be placed anywhere within the definition specifications, a subprocedure must be coded in the order shown above.

You cannot code the following for subprocedures:
- Prerun-time and compile-time arrays and tables
- *DTAARA definitions

The calculation specifications are processed only once and the procedure returns at the end of the calculation specifications. See “Subprocedure Calculations” on page 67 for more information.

A subprocedure may be exported, meaning that procedures in other modules in the program can call it. To indicate that it is to be exported, specify the keyword EXPORT on the Procedure-Begin specification. If not specified, the subprocedure can only be called from within the module. Note that procedures can be exported only from NOMAIN DLLs.

Procedure Interface Definition

If a prototyped procedure has call parameters or a return value, then it must have a procedure interface definition. A **procedure interface definition** is a repetition of the prototype information within the definition of a procedure. It is used to declare the entry parameters for the procedure and to ensure that the internal definition of the procedure is consistent with the external definition (the prototype).

You specify a procedure interface by placing PI in the Definition-Type entry (positions 24-25). Any parameter definitions, indicated by blanks in positions 24-25, must immediately follow the PI specification. The procedure interface definition ends with the first definition specification with non-blanks in positions 24-25 or by a non-definition specification.

For more information on procedure interface definitions, see “Procedure Interface” on page 75.

Return Values

A procedure that returns a value is essentially a user-defined function, similar to a built-in function. To define a return value for a subprocedure, you must:

1. Define the return value on both the prototype and procedure-interface definitions of the subprocedure.
2. Code a RETURN operation with an expression in the extended-factor 2 field that contains the value to be returned.

You define the length and the type of the return value on the procedure-interface specification (the definition specification with PI in positions 24-25). The following keywords are also allowed:

DATFMT(fmt)
The return value has the date format specified by the keyword.

DIM(N)
The return value is an array with N elements.

LIKE(name)
The return value is defined like the item specified by the keyword.

LIKEDS(name)
The return value is a data structure defined like the data structure specified by the keyword.

LIKEREC(name{,type})
The return value is a data structure defined like the record name specified by the keyword.

PROCPTR
The return value is a procedure pointer.

TIMFMT(fmt)
The return value has the time format specified by the keyword.

To return the value to the caller, you must code a RETURN operation with an expression containing the return value. The expression in the extended-factor 2 field is subject to the same rules as an expression with EVAL. The actual returned value has the same role as the left-hand side of the EVAL expression, while the extended factor 2 of the RETURN operation has the same role as the right-hand side. You must ensure that a RETURN operation is performed if the subprocedure has a return value defined; otherwise an exception is issued to the caller of the subprocedure.

Scope of Definitions

Any items defined within a subprocedure are local. If a local item is defined with the same name as a global data item, then any references to that name inside the subprocedure use the local definition.

However, keep in mind the following:
- Subroutine names and tag names are known only to the procedure in which they are defined, even those defined in the main procedure of an EXE.
- All fields specified on input and output specifications are global. When a subprocedure uses input or output specifications (for example, while processing a read operation), the global name is used even if there is a local variable of the same name.

When using a global KLIST or PLIST in a subprocedure some of the fields may have the same names as local fields. If this occurs, the global field is used. This may cause problems when setting up a KLIST or PLIST prior to using it.

For example, consider the following source:

```
D* Main procedure definitions
D Fld1            S              1A
D Fld2            S              1A
D*
C* Define a global key field list with 2 fields, Fld1 and Fld2
C     global_kl     KLIST
C                   KFLD                    Fld1
C                   KFLD                    Fld2
C*
P* Subprocedure Section
P Subproc         B
D Fld2            S              1A
D*
C* local_kl has one global kfld (fld1) and one local (fld2)
C*
C     local_kl      KLIST
C                   KFLD                    Fld1
C                   KFLD                    Fld2
C*
C* Even though Fld2 is defined locally in the subprocedure,
C* the global Fld2 is used by the global_kl, since global KLISTs
C* always use global fields.  As a result, the assignment to the
C* local Fld2 will NOT affect the CHAIN operation.
C*
C                   EVAL      Fld1 = 'A'
C                   EVAL      Fld2 = 'B'
C     global_kl     SETLL     file
C*
C* Local KLISTs use global fields only when there is no local
C* field of that name.  local_kl uses the local Fld2 and so the
```

```
C* assignment to the local Fld2 WILL affect the CHAIN operation.
C                   EVAL      Fld1 = 'A'
C                   EVAL      Fld2 = 'B'
C     local_kl      SETLL     file
 ...
P                 E
```

For more information on the placement of definitions and their effect on scope, see "Placement of Definitions and Scope" on page 256.

Subprocedure Calculations

A subprocedure ends when one of the following occurs:

- A RETURN operation is processed.
- The last calculation in the body of the subprocedure is processed.

Figure 15 on page 68 shows the normal processing steps for a subprocedure. Figure 16 on page 69 shows the exception/error handling sequence.

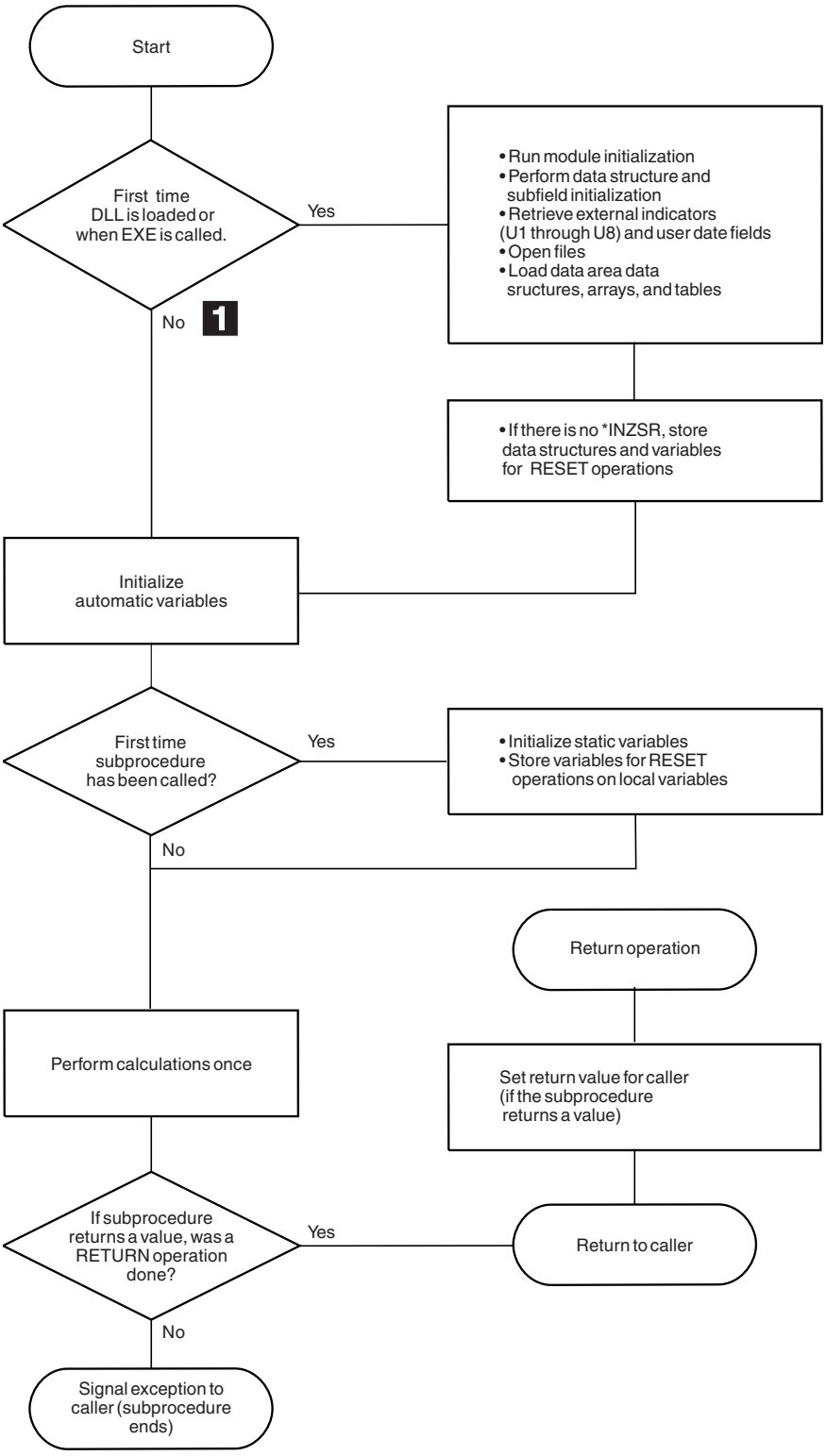


Figure 15. Normal Processing Sequence for a Subprocedure

1 Taking the "No" branch means that another procedure has already been called since the program was activated. You should ensure that you do not make any incorrect assumptions about the state of files, data areas, etc., since another procedure may have closed files, or unlocked data areas.

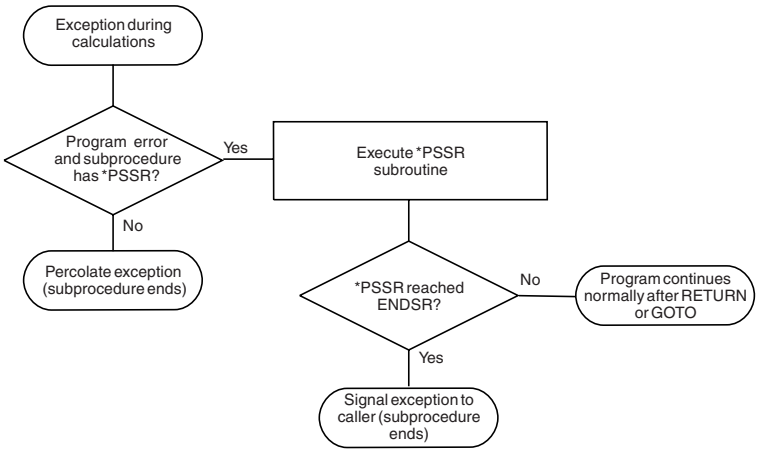


Figure 16. Exception/Error Handling for a Subprocedure

Here are some points to consider when coding subprocedures:

- There is no *INZSR associated with subprocedures. Data is initialized (with either INZ values or default values) when the subprocedure is first called, but before the calculations begin.
- When a subprocedure returns normally, the return value, if specified on the prototype of the called program or procedure, is passed to the caller. Nothing else occurs automatically. All files and data areas must be closed manually. Files must be written out manually. In the case of an EXE, you can set on indicators such as LR, but program termination will not occur until the main procedure for the EXE terminates.
- Exception handling within a subprocedure differs from a main procedure primarily because there is no default exception handler for subprocedures and so situations where the default handler would be called for a main procedure correspond to abnormal end of the subprocedure. For example, Factor 2 of an ENDSR operation for a *PSSR subroutine within a subprocedure must be blank. A blank factor 2 normally would result in control being passed to the default handler, but in a subprocedure, if the ENDSR is reached, then the subprocedure will end abnormally.

  You can avoid abnormal termination either by coding a RETURN operation in the *PSSR, or by coding a GOTO and label in the subprocedure to continue processing.
- The *PSSR error subroutine is local to the subprocedure. Conversely, file errors are global by definition, and so you cannot code an INFSR in a subprocedure, nor can you use a file for which an INFSR is coded.

NOMAIN Module

You can code one or more subprocedures in a module without coding any action subroutines. Such a module is called a **NOMAIN module**, and it requires the specification of the NOMAIN keyword on the control specification. The concept of a NOMAIN DLL is similar to that of an OS/400™ service program.

For NOMAIN DLLs, the following should be considered:

- The DLL must consist of procedures only. All subroutines (BEGSR) must be local to a procedure.
- No GUI operation codes allowed in the source. These include START, STOP, SETATR, GETATR, %SETATR, %GETATR, SHOWWIN, CLSWIN, and READS.

DSPLY can be used. However, if the procedure containing it is called from a VisualAge RPG DLL, then the DSPLY operation code does nothing.
- *INZSR and *TERMSR are not permitted.
- *ENTRY parameters are not permitted.

EXE Module

A module is called an **EXE module**, since it requires the specification of the EXE keyword on the control specification.

The EXE module consists of a main procedure and subprocedures. All subroutines (BEGSR) must be local to a procedure. The EXE must contain a procedure whose name matches the name of the source file. This will be the main entry point for the EXE, that is, the main procedure.

For EXE modules, the following should be considered:
- No GUI operation codes are allowed in the source. This includes START, STOP, SETATR, GETATR, %SETATR, %GETATR, SHOWWIN, CLSWIN and READS. DSPLY can be used.
- *INZSR and *TERMSR are not permitted.
- *ENTRY parms are not permitted.

  If there are entry parameters, they are specified on the parameter definition for the main procedure, and they must be passed in by VALUE (the VALUE keyword must be specified for each parameter).They cannot be UCS-2 parameters.
- The EXPORT keyword is not allowed on the Begin P specification.
- The return value for the main procedure must be defined as a binary or integer of precision zero(0).

Subprocedures and Subroutines

A subprocedure is similar to a subroutine, except that a subprocedure offers the following improvements:
- You can pass parameters to a subprocedure, even passing by value.

  This means that the parameters used to communicate with subprocedures do not have to be modifiable. Parameters that are passed by reference, as they are with programs, must be modifiable, and so may be less reliable.
- The parameters passed to a subprocedure and those received by it are checked at compile time for consistency. This helps to reduce run-time errors, which can be more costly.
- You can use a subprocedure like a built-in function in an expression.

  When used in this way, they return a value to the caller. This basically allows you to custom-define any operators you might need in an expression.
- Names defined in a subprocedure are not visible outside the subprocedure.

  This means that there is less chance of the procedure inadvertently changing a item that is shared by other procedures. Furthermore, the caller of the procedure does not need to know as much about the items used inside the subprocedure.
- You can call the subprocedure from outside the module, if it is exported.
- You can call subprocedures recursively.
- Procedures are defined on a different specification type, namely, procedure specifications. This different type helps you to immediately recognize that you are dealing with a separate unit.

Nonetheless, if you do not require the improvements offered by subprocedures, you should use a subroutine. The processing of a subroutine is much faster than a call to a subprocedure.

Prototypes and Parameters

The recommended way to call programs and procedures is to use prototyped calls, since prototyped calls allow the compiler to check the call interface at compile time. If you are coding a subprocedure, you will need to code a procedure-interface definition to allow the compiler to match the call interface to the subprocedure.

This section describes how to define each of the following: prototypes, prototyped parameters, and procedure-interface definitions.

Prototypes

A **prototype** is a definition of the call interface. It includes the following information:

- Whether the call is bound (procedure) or dynamic (program)
- How to find the program or procedure (the external name)
- If it is a remote program residing on an iSeries server
- The number and nature of the parameters
- Which parameters must be passed, and which are optionally passed
- The data type of the return value, if any (for a procedure)

A prototype must be included in the definition specifications of the program or procedure that makes the call. The prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters.

The following rules apply to prototype definitions.

- A prototype name must be specified in positions 7-21. If the keyword EXTPGM or EXTPROC is specified on the prototype definition, then any calls to the program or procedure use the external name specified for that keyword. If neither keyword is specified, then the external name is the prototype name, that is, the name specified in positions 7-21 (in uppercase).
- Specify PR in the Definition-Type entry (positions 24-25). Any parameter definitions must immediately follow the PR specification. The prototype definition ends with the first definition specification with non-blanks in positions 24-25 or by a non-definition specification.
- Specify any of the following keywords as they pertain to the call interface:

  EXTPROC(name)
  The call will be a bound procedure call that uses the external name specified by the keyword.

  EXTPGM(name)
  Together with LINKAGE(*SERVER), determines the call will be to a remote program on an iSeries server using the external program name specified by the keyword.

  CLTPGM(name)
  The call will be an external program call that uses the external name specified by the keyword.

DLL(name)
The DLL keyword, together with the LINKAGE keyword, is used to prototype a procedure that calls functions in Windows DLLs, including Windows APIs.

LINKAGE(name | *SERVER)
The LINKAGE keyword, together with the DLL keyword, specifies the Linkage convention (interface) to be used when invoking functions in a DLL.

LINKAGE(*SERVER), together with the EXTPGM keyword, specify the prototype is for a remote program on an iSeries server.

STATIC
The STATIC keyword specifies that the data item is to be stored in static storage, and thereby hold its value across calls to the procedure in which it is defined.

- A return value, if any, is specified on the PR definition. Specify the length and data type of the return value. In addition, you may specify the following keywords for the return value:

  DATFMT(fmt)
  The return value has the date format specified by the keyword.

  DIM(N)
  The return value is an array or data structure with N elements.

  LIKE(name)
  The return value is defined like the item specified by the keyword.

  LIKEDS(data_structure_name)
  The returned value is a data structure. (You cannot refer to the subfields of the return value when you call the procedure.)

  LIKEREC(name{,type})
  The returned value is a data structure defined like the specified record format name.

  Note: You cannot refer to the subfields of the return value when you call the procedure.

  PROCPTR
  The return value is a procedure pointer.

  TIMFMT(fmt)
  The return value has the time format specified by the keyword.

  VARYING
  A character, graphic, or UCS-2 return value has a variable-length format.

For information on these keywords, see “Definition-Specification Keywords” on page 264.

Figure 17 on page 73 shows a prototype for a subprocedure CVTCHR that takes a numeric input parameter and returns a character string. Note that there is no name associated with the return value. For this reason, you cannot display its contents when debugging the program.

```
     * The returned value is the character representation of
     * the input parameter NUM, left-justified and padded on
     * the right with blanks.
 *
    D CVTCHR          PR            31A
    D   NUM                         31P 0   VALUE
 *
     * The following expression shows a call to CVTCHR.  If
     * variable rrn has the value 431, then after this EVAL,
     * variable msg would have the value
     *    'Record 431 was not found.'
 *
    C                   EVAL      msg = 'Record '
    C                                 + %TRIMR(CVTCHR(RRN))
    C                                 + ' was not found '
```

Figure 17. Prototype for CVTCHR

If you are writing a prototype for an exported subprocedure or for a main procedure, put the prototype in a /COPY file and copy the prototype into the source file for both the callers and the module that defines the procedure. This coding technique provides maximum parameter-checking benefits for both the callers and the procedure itself, since they all use the same prototype.

Prototyped Parameters

If the prototyped call interface involves the passing of parameters, then you must define the parameter immediately following the PR specification. The following keywords, which apply to defining the type, are allowed on the parameter definition specifications:

ASCEND
The array is in ascending sequence.

CCSID(number | *DFT)
Sets the CCSID for graphic and UCS-2 definitions.

CLASS(*JAVA:class_name)
For Java only, provides the class of the object for fields that can store objects.

DATFMT(fmt)
The date parameter has the format fmt.

DIM(N)
The parameter is an array or data structure with N elements.

LIKE(name)
The parameter is defined like the item specified by the keyword.

LIKEREC(name{,type})
The parameter is a data structures whose subfields are the same as the fields in the specified record format name.

LIKEDS(data_structure_name)
The parameter is a data structure whose subfields are the same as the subfields identified in the LIKEDS keyword.

PROCPTR
The parameter is a procedure pointer.

TIMFMT(fmt)
The time parameter has the format fmt.

VARYING
A character, graphic, or UCS-2 return value has a variable-length format.

For information on these keywords, see "Definition-Specification Keywords" on page 264.

The following keywords, which specify how the parameter should be passed, are also allowed on the parameter definition specifications:

CONST
The parameter is passed by read-only reference. A parameter defined with CONST must not be modified by the called program or procedure. This parameter-passing method allows you to pass literals and expressions.

NOOPT
The parameter will not be optimized in the called program or procedure.

OPTIONS(opt1 { : opt2 { : opt3 { : opt4 } } })
Where opt1 ... opt4 can be the following parameter passing options:

***HEX** Valid only for remote program calls, this indicates the parameter should be passed as if it were a hex value, without the automatic data conversion between the Windows client and iSeries server.

***OMIT** The special value *OMIT may be passed for this reference parameter.

***VARSIZE** The parameter may contain less data than indicated on the definition. This keyword is valid only for character parameters, graphic parameters, or arrays passed by reference. The called program or procedure must have some way of determining the length of the passed parameter.

Note: When this keyword is omitted for fixed-length fields, the parameter may only contain more or the same amount of data as indicated on the definition; for variable-length fields, the parameter must have the same declared maximum length as indicated on the definition.

***RIGHTADJ** For a CONST or VALUE parameter, *RIGHTADJ indicates that the graphic, UCS-2, or character parameter value is to be right adjusted.

***STRING** Pass a character value as a null-terminated string. This keyword is valid only for basing pointer parameters passed by a value or by read-only reference.

***TRIM** The parameter is trimmed before it is passed. This option is valid for character, UCS-2 or graphic parameters passed by value or by read-only reference. It is also valid for pointer parameters that have OPTIONS(*STRING) coded.

Note: When a pointer parameter has OPTIONS(*STRING : *TRIM) specified, the value will be trimmed even if a pointer is passed directly. The null-terminated string that the pointer is pointing to will be copied into a temporary, trimmed of blanks, with a new

null-terminator added at the end, and the address of that temporary will be passed.

VALUE
The parameter is passed by value.

For information on the keywords listed above, see "Definition-Specification Keywords" on page 264.

Procedure Interface

If a prototyped procedure has call parameters or a return value, then a procedure interface definition must be defined, either in the main source section (for a main procedure) or in the subprocedure section. A **procedure interface definition** repeats the prototype information within the definition of a procedure. It is used to declare the entry parameters for the procedure and to ensure that the internal definition of the procedure is consistent with the external definition (the prototype).

The following rules apply to procedure interface definitions:
- The name of the procedure interface, specified in positions 7-21, is optional. If specified, it must match the name specified in positions 7-21 on the corresponding prototype definition.
- Specify PI in the Definition-Type entry (positions 24-25). The procedure-interface definition can be specified anywhere in the definition specifications. In the main procedure, the procedure interface must be preceded by the prototype that it refers to. A procedure interface is required in a subprocedure if the procedure returns a value, or if it has any parameters; otherwise, it is optional.
- Any parameter definitions, indicated by blanks in positions 24-25, must immediately follow the PI specification.
- Parameter names must be specified, although they do not have to match the names specified on the prototype.
- All attributes of the parameters, including data type, length, and dimension, must match exactly those on the corresponding prototype definition.
- To indicate that a parameter is a data structure, use the LIKEDS keyword to define the parameter with the same subfields as another data structure.
- The keywords specified on the PI specification and the parameter specifications must match those specified on the prototype.

If a module contains calls to a procedure, then there must be a prototype definition for each program and procedure that you want to call. One way of minimizing the required coding is to store shared prototypes in /COPY files.

If you provide prototyped procedures to other users, be sure to provide them with the prototypes (in /COPY files) as well.

Chapter 7. SQL Support

If your VisualAge RPG application contains Structured Language (SQL) statements to access DB2® databases, you must perform the following tasks:

1. Install DB2 and set up access to it. The DB2 manuals *DB2 Universal Database Personal Edition Quick Beginnings*, S10J-8150 and *DB2 Universal Database for Windows NT Quick Beginnings*, S10J-8149, and the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** (at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**) describe how to install and setup the DB2 products on workstation and iSeries servers.
2. Code the SQL statements in your source program. "General Syntax Rules" describes how to code SQL statement in a VisualAge RPG program.
3. Build the application. The online help for the Build Options dialog from the GUI Designer describes which build options can be selected for VisualAge RPG programs with SQL statements. For additional information on building, running, and connecting to databases, see "Building an Application" on page 85, "Running an Application" on page 86, and "Connecting to a Database" on page 86.
4. Package and install the user application. *Programming with VisualAge RPG*, SC09-2449-05 describes how to package and install a VisualAge RPG application.

The VisualAge RPG embedded SQL support differs from most other implementations in that there is no separate precompiler for creating the intermediate file which is then compiled. The embedded SQL statements are handled during the compile step of the build process.

Your application can be built to use local databases, databases on other workstation nodes, or databases on other iSeries servers. Any differences in the level of SQL supported on these other systems are overlooked with the level of SQL supported on the workstation where the build is performed. Only the syntax supported by DB2 on the build-time workstation is allowed.

If you port your application to another workstation, the application can only run on the same or higher level of DB2.

Note: VisualAge RPG supports the level of function defined in DB2/2 V1.2. More recent releases of DB2/2 can be used if only V1.2 functions are used in the application.

General Syntax Rules

The following rules describe the syntax for SQL statements which are included in your VisualAge RPG source program:

1. SQL statements are coded in the calculation specifications. The following statements are exceptions and can be coded anywhere before any compile-time data (** in positions 1 to 2): INCLUDE, BEGIN DECLARE, END DECLARE.
2. To specify the beginning of an SQL statement, code **/EXEC SQL** in positions 7 to 15. Position 16 must be blank. The remainder of the line from positions 17 to 80 can either be an SQL statement or part of an SQL statement.
3. To specify the end of an SQL statement, code **/END-EXEC** in positions 7 to 15.
4. Only one SQL statement can be coded between /EXEC SQL and /END-EXEC.

5. An SQL statement can be coded on several lines. The lines between the /EXEC SQL and the /END-EXEC must contain a plus sign (+) in position 7 and a blank in position 8.
6. Character literals can span several lines. The literal is coded up to column 80 on one line and continues on column 9 of the next line.
7. To specify a comment line within the SQL statement, code an asterisk (*) in position 7.
8. To specify a comment on the same line as an SQL statement, use -- within the SQL statement.
9. Names beginning with SQL should be avoided in the program since they may conflict with SQL names.

The following example illustrates the general syntax rules:

```
----+-*--1----+----2----+----3----+----4----+----5----+----6----+
     C/EXEC SQL WHENEVER SQLERROR GO TO ERRLAB
     C/END-EXEC
     C/EXEC SQL                  -- starts SQL statement
     C+   SELECT *
      * this is a normal RPG style comment
     C+      INTO :hvar1,        -- host variable one
     C+           :hvar2         -- host variable two
     C+      FROM TABLEX
     C+      WHERE NAME='TESTING'
     C/END-EXEC
```

Figure 18. General Syntax Rules for SQL Statements

Host Variable Declarations

The SQL statements BEGIN DECLARE and END DECLARE are allowed in VisualAge RPG programs, however these statements are ignored. All variables that are declared are considered candidate host variables.

Host variables are identified by a preceding colon in an SQL statement.

The data types supported for host variables are character, variable-length character, graphic, integer packed decimal, zoned decimal, binary numeric, date, time, and timestamp. The SQL data types REAL, DOUBLE and VARCHAR are not supported.

The following table summarizes how VisualAge RPG data types map to SQL data types.

Table 11. Host Data Types

VARPG Data Type	SQL Data Type	Description	Notes
4 digit binary	SMALLINT	16 bit signed integer	No decimal positions
9 digit binary	INTEGER	32 bit signed integer	No decimal positions
Packed decimal	DECIMAL(m,n)	Default RPG numeric data type	
Character	CHAR(m)	Fixed length character	Up to 254 chars
Graphic	GRAPHIC(m)	Fixed length DBCS string	Up to 127 chars
Zoned decimal	DECIMAL(m,n)	Fixed point number	Converted to packed decimal by RPG before or after DB2 operation.
Date	DATE	Date	The following formats are supported by DB2: *ISO, *USA, *EUR, *JIS. Other RPG formats are converted by RPG before or after DB2 operation.
Time	TIME	Time	The following formats are supported by DB2: *ISO, *USA, *EUR, *JIS. Other RPG formats are converted by RPG before or after DB2 operation.
Timestamp	TIMESTAMP	Timestamp	

Host Variable Rules

The following describes the rules for host variables:

1. A host variable may be any scalar character, numeric, date, time, timestamp, or DBCS field defined in the program. A host variable cannot be any of the following:
   - Multiple occurrence data structures
   - Indicator field names (*INxx)
   - Tables
   - UDATE, UDAY, UMONTH, UYEAR
2. Indexed arrays are not allowed as host variables.
3. All numeric data types in SQL are compatible and the appropriate conversions occur when the host variable type does not match the column definition. This includes database columns in scientific notation (FLOAT). You will receive a message indicating truncation.
4. All character data types in SQL are compatible and the appropriate conversions occur when the host variable type does not match exactly the column definition. SQL will perform the appropriate conversions between fixed length and varying length character. You will receive a message indicating truncation.
5. All DBCS data types in SQL are compatible and the appropriate conversions occur when the host variable type does not match exactly the column definition. SQL will perform the appropriate conversions between fixed length and varying length DBCS data. You will receive a message indicating truncation.
6. Date, time, and timestamp fields in SQL are compatible with character fields. For example, when an SQL date column is fetched into a character host variable, it is formatted using the Date/Time format value specified on the DB2 options page of the Build notebook.
7. Indicator variables must be declared as 4 digit binary numeric.
8. Single occurrence data structures with no subfields are considered character data type following normal RPG rules (see Chapter 11, “Data Structures,” on page 157). Data structures with subfields are considered host structures.

Data Structures as Host Variables

When a data structure is specified as a host variable in a SQL statement, the name refers to all subfields of the data structure. This is a convenient way to specify a long list of host variables. Figure 19 illustrates the source code if a data structure is not used while Figure 20 illustrates the source code if a data structure is used.

```
----+-*--1----+----2----+----3----+----4----+----5----+----6----+
     C/EXEC SQL
     C+   SELECT *
     C+      INTO :F1, :F2, :F3, :F4, :F5, :F6, :F7
     C+      FROM TABLEX
     C+      WHERE NAME='GASPARE'
     C/END-EXEC
```

Figure 19. Coding host variables without using a data structure

Using data structure:

```
----+-*--1----+----2----+----3----+----4----+----5----+----6----+
     D ROW             DS
     D   F1                    1     10
     D   F2                   11     20
     D   F3                   21     30
     D   F4                   31     40
     D   F5                   41     50
     D   F6                   51     60
     D   F7                   61     70
      *
     C/EXEC SQL
     C+   SELECT *
     C+      INTO :ROW
     C+      FROM TABLEX
     C+      WHERE NAME='GASPARE'
     C/END-EXEC
```

Figure 20. Coding host variables using a data structure

Although there is some extra coding for the data structure subfields, the host variable list in the SQL statement is much smaller. Since there are likely to be several SQL statements in the program, the overall coding effort can be less.

Indicator Variables and Structures

If indicator variables are required (for example, by null columns), then a short binary numeric array can be specified along with the name of the host structure.

```
----+-*--1----+----2----+----3----+----4----+----5----+----6----+
     D ROW             DS
     D   F1                     1     10
     D   F2                    11     20
     D   F3                    21     30
     D   F4                    31     40
     D   F5                    41     50
     D   F6                    51     60
     D   F7                    61     70
      *
     D STRUCT          DS
     D  AI                      1     14B 0 DIM(7)
      *
     C/EXEC SQL
     C+   SELECT *
     C+      INTO :ROW:AI
     C+      FROM TABLEX
     C+      WHERE NAME='GASPARE'
     C/END-EXEC
```

Figure 21. Indicator variables and structures

This is the same as to coding each array element as an indicator variable. For example, the indicator variable for field F1 is AI(1); for field F2, AI(2); etc.

Host Structure Rules

The following describes the rules for host structures:

- A single SQL statement can contain one or more host structures.
- The data structure must contain subfields in order to be recognized as a host structure. A data structure without subfields is considered a normal character field.
- A host structure name can be followed immediately by an indicator array, which is a binary numeric array with zero decimal positions. Each element of the array corresponds to a subfield of the data structure.

/EXEC SQL INCLUDE Statement

The /EXEC SQL INCLUDE statement can appear anywhere in the program prior to the compile time data section (** in positions 1-2). The filename is specified by a single name. The file extension defaults to VPG.

Note: The filename can only refer to a local file.

/EXEC SQL INCLUDE SQLCA Statement

An SQLCA data structure is automatically included in the VisualAge RPG program when database processing has been specified on the DB2 options page of the Build notebook. The data structure is included even if the INCLUDE SQLCA statement is not specified.

You can use the SQLCA data structure to query the result of each SQL statement after it has been executed.

If the INCLUDE SQLCA statement is specified, the definition for the data structure is included at that point in the program. Subsequent instances of the INCLUDE SQLCA statement are ignored.

Note: The SQLCA data structure can also be included using the /COPY compiler directive, instead of /EXEC SQL INCLUDE SQLCA.

Figure 22 shows the layout of the SQLCA data structure:

```
----+-*--1----+----2----+----3----+----4----+----5----+----6----+
SQL  D* Start of SQLCA Data Structure
SQL  D SQLCA           DS
SQL  D  SQLAID                  1      8A
SQL  D  SQLABC                  9     12B 0
SQL  D  SQLCOD                 13     16B 0
SQL  D  SQLERL                 17     18B 0
SQL  D  SQLERM                 19     88A
SQL  D  SQLERP                 19     96A
SQL  D  SQLERRD                97    120B 0 DIM(6)
SQL  D  SQLERR                 97    120A
SQL  D   SQLER1                97    100B 0
SQL  D   SQLER2               101    104B 0
SQL  D   SQLER3               105    108B 0
SQL  D   SQLER4               109    112B 0
SQL  D   SQLER5               113    116B 0
SQL  D   SQLER6               117    120B 0
SQL  D  SQLWRN                121    127A
SQL  D   SQLWN0               121    121A
SQL  D   SQLWN1               122    122A
SQL  D   SQLWN2               123    123A
SQL  D   SQLWN3               124    124A
SQL  D   SQLWN4               125    125A
SQL  D   SQLWN5               126    126A
SQL  D   SQLWN6               127    127A
SQL  D   SQLWN7               128    128A
SQL  D   SQLWN8               129    129A
SQL  D   SQLWN9               130    130A
SQL  D   SQLWNA               131    131A
SQL  D  SQLSTT                132    136A
SQL  D* End of SQLCA Data Structure
```

Figure 22. Source expansion for SQLCA data structure

/EXEC SQL WHENEVER Statement

The /EXEC SQL WHENEVER statement determines what error handling is done following execution of SQL statements. Figure 23 illustrates the syntax of the /EXEC SQL WHENEVER statement.

```
C/EXEC SQL WHENEVER <condition> <action>
C/END-EXEC
```

Figure 23. Syntax of SQL WHENEVER statement

Note: <condition> is SQLWARNING, SQLERROR, or NOT FOUND. <action> is GOTO <tag-name>, GO TO <tag-name>, or CONTINUE.

The /EXEC SQL WHENEVER identifies the action to be performed when an SQL statement returns with a non-zero return code. It applies to all subsequent SQL statements in the program up to the next /EXEC SQL WHENEVER statement.

A message is issued whenever the action is inapplicable based on the section of code that the statement is in. Figure 24 illustrates this.

```
---+-*--1----+----2----+----3----+----4----+----5----+----6----+
 1  C           SUBR1     BEGSR
 2  C/EXEC SQL WHENEVER SQLERROR GOTO ERRLAB
 3  C/END-EXEC
 4  C           ERRLAB    TAG
 5  C                     ENDSR
 6  C           SUBR2     BEGSR
 7  C/EXEC SQL FETCH ...
 8  C/END-EXEC
 9  C                     ENDSR
```

Figure 24. Error messages using SQL WHENEVER

In this example, statement 7 is invalid since the WHENEVER action would cause a branch into another subroutine. The possible values for the condition are:

- SQLWARNING: The action is invoked if the value of the SQL return code is greater than 0 and less than 100.
- SQLERROR: The action is invoked if the value of the SQL return code is less than 0.
- NOT FOUND: The action is invoked if the value of the SQL return code is 100.

The possible values for the action are:

- GOTO <tag-name>: If the condition is true, execution resumes at the specified tag name.
- CONTINUE: If the condition is true, execution resumes at the next executable statement is the program. This is the default action.

Note: The /EXEC SQL WHENEVER statement must appear in the calculation specifications.

/EXEC SQL BEGIN DECLARE Statement

The /EXEC SQL END DECLARE statement are ignored by the compiler, however the statements in between are not ignored. Table 12 describes how SQL data types map to VisualAge RPG data types.

Table 12. Mapping of SQL types to host variables

SQL Data Type	VARPG Data Type	Data Format (pos 43)	Length (Bytes) (pos 44-51)	Decimal Positions (pos 52)
SMALLINT	4 digit binary	B	2	0
INTEGER	9 digit binary	B	4	0
DECIMAL(m,n)	Packed decimal	P	m/2+1	n
CHAR(m)	Character		m	
DATE	Date		10	
TIME	Time		8	
TIMESTAMP	Timestamp		26	
GRAPHIC(m)	Graphic	G	m*2	

Runtime Error Handling

If an SQL statement fails, no messages are issued during run time. You must code an SQL WHENEVER statement or explicitly check the SQLCOD value in order to detect these errors.

Building an Application

To build an application that contains embedded SQL, you must specify the following options on the Build notebook:

- DB2 database name
- Either a Package name or a Bind file name.

For more information, see *Programming with VisualAge RPG*, SC09-2449-05.

The database name you specify must be cataloged on your workstation. You must have the proper authority to use the database. When you start building an application, the DB2 Database Manager is started automatically (the build process issues the DB2START command). However, if you are building your application from a client environment, you must start the database manager yourself on the server. To connect to the database automatically during compilation, you must first specify a valid userid and password on the DB2connect page of the Build Options Notebook. For subsequent builds, VARPG will use this information to connect to the database.

Before your application can be run, a package must be created. A package is an object stored in the database that includes information to execute the embedded SQL in your application, or program. If the package is created at build time, this is called binding enabled. This allows the application to only access the database used during the build. If the application is built with binding deferred, a bind file is created and the application can access many databases.

Running an Application

To run an application that contains embedded SQL, the following conditions must exist:

- The database that your application accesses must be cataloged on your workstation
- You must have the proper authority to access the database
- The timestamp with which your application was built must match the timestamp of the database package you are accessing.

When you run your VisualAge RPG application, the DB2 Database Manager is started automatically (DB2START). If you are running your application from a client environment, you must start the database manager yourself on the server.

If the application is built with binding deferred, the bind files that are produced must be bound to the database before the application can run.

When you build your application, a timestamp is embedded in it. This timestamp is compared to the database package when the application is run. If the timestamps are not equal, the application will not run. This mismatch can occur if the application is run against an older package.

If you port your application to another workstation, the timestamps in your application must match the timestamps in the package that you are accessing on the new workstation. You can either:

- Rebind your application by issuing the following command from a command prompt:

  ```
  sqlbind applic1.bnd typesx
  ```

  where *sqlbind* is the DB2 command, *applic1.bnd* is the bind file created during the build, and *typesx* is the database you wish to access.
- Setup access to the database used by your application. You must catalog the database that you are trying to access on the new workstation. This is the same database used during the build. The database can be on another workstation, or on a remote system.

Connecting to a Database

Before your application can access a database, your application must have a connection to the database. You can do this by either using the CONNECT TO statement or by using an implicit connect.

Using the CONNECT TO Statement

You can specify the database name you wish to connect to by using the CONNECT TO statement in your application. For example,

```
C\EXEC SQL CONNECT TO LATONA
C\END-EXEC
```

Note: LATONA is the name of the database.

You can use a variable for the database name as shown in the following example:

```
D server          s             10a
D userid          s              8a
D password        s             10a
...
C                   eval      server = 'LATONA'
```

```
C                   eval      userid = 'USERID'
C                   eval      password = 'password'
...
C\EXEC SQL
C+     CONNECT TO :server IN SHARE MODE user :userid using :password
C\END-EXEC
```

For more information on the syntax of the CONNECT SQL statement, refer to the SQL Reference for your DB2 configuration.

Using an Implicit Connect

You can establish an implicit connection to your database by setting the environment variable SQLDBDFT to point to the database that you want to implicitly connect to. For example,

```
SET SQLDBDFT=LATONA
```

This environment variable can be set either in your CONFIG.SYS file, or set from the session's command prompt.

If you are running your application in a Windows environment, you can use the following to connect to a database:

```
SET DB2DBDFT=LATONA
SET DB2USERID=USERID
SET DB2PASSWORD=password
```

These environment variables are set in the AUTOEXEC.BAT file.

Note: Some differences exist in the environment variable names depending on the configuration of DB2 installed. Refer to the DB2 installation manuals.

Chapter 8. File Considerations

This section describes how to use files in a VisualAge RPG program. Your program can use DISK, PRINTER, and SPECIAL files:

- DISK files:
  - DISK files can either be remote or local
  - Remote DISK files must be externally described
  - Local DISK files must be program described
- PRINTER files:
  - A maximum of eight PRINTER files are allowed
  - PRINTER files must be program described
  - PRINTER files must be local
- SPECIAL files:
  - SPECIAL files must be program described
  - SPECIAL files must be local

For more information on how to specify files, see the following sections:

- Chapter 17, "File Description Specifications," on page 237
- Chapter 18, "Definition Specifications," on page 255
- Chapter 19, "Input Specifications," on page 299
- Chapter 21, "Output Specifications," on page 321.

Disk Files

Part 3, "Specifications," on page 209 describes how to define both local and remote files on the various specifications. This section describes additional considerations if your VisualAge RPG application uses local files or OS/400 database files.

Local Files

The following is a summary of restrictions for local files:

- Local file cannot be locked.
- Numeric fields in local files are written and read as is, with no conversion.
- If your program performs I/O on a local file does not exist, the file is created.
- If the local file is created by VisualAge RPG, records in this file are terminated with a carriage return line feed. If you use a local file that does not contain carriage return line feeds, your VisualAge RPG application will not be able to perform I/O operations on this file.
- Bit patterns in a local file are read into storage as is. If a bit pattern contains the binary representation for a carriage return line feed (CRLF), the record that is read by your VisualAge RPG program will be split into two records.
- If a local file contains binary numbers, the numbers are byte-swapped.

OS/400 Files

VisualAge RPG operation codes can access OS/400 physical, source physical, and logical database files. For more information on how to setup OS/400 database files, see the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**.

Before your application can access OS/400 database files, you must set up the server. For more information on how to setup a server, see *Getting Started with WebSphere Development Studio Client for iSeries*, SC09-2625-06 and *Programming with VisualAge RPG*, SC09-2449-05.

When your VisualAge RPG application accesses an OS/400 database file, a DDM service job is used to handle the database I/O requests. A single DDM service job is used on each different iSeries server where files are opened for each VisualAge RPG application. When the application ends, all the DDM service jobs end.

Sharing the File Open Data Path

Sharing open data paths is not supported. If a VisualAge RPG application contains an OPNQRYF CL command, then the open data path associated with the OPNQRYF command cannot be shared by files opened in the VisualAge RPG application.

If the VisualAge RPG application calls a program which uses the OPNQRYF command, the open data path can be shared.

Each open performed by the VisualAge RPG application creates a unique open data path. Multiple opens of the same database file within the same VisualAge RPG application result in different instances of the file, each with its own open data path.

You can open the same database file more than once by using different file alias names in the VisualAge RPG program to refer to the same actual OS/400 database file. You must define multiple file pages in the Define iSeries Information notebook.

You can also open the same database file in different components of the VisualAge RPG application.

Query Files and Single/Blocked Record I/O Operations

When using query files (OPNQRYF command), set both the VisualAge RPG application and the query file to expect either single or blocked record input/output operations. If these settings do not match, the application appears to skip records on input. This mismatch can occur because the OPNQRYF command first opens the file, then the open data path is shared with the VisualAge RPG application file open request, ignoring some of its open settings. However, the single or blocked record setting is fixed in the VisualAge RPG application at compile time.

For the OPNQRYF and OVRDBF commands, the SEQONLY(*NO/*YES) keyword determines single or blocked record input or output processing. For VisualAge RPG applications, single or blocked record processing is based on one of the following:

- The F specification keyword BLOCK(*NO/*YES) value
- The presence of random record positioning operations (SETLL, for example) on the file in the program.

Invalid Data Errors on Query Files

A run time error can occur if a query file (OPNQRYF command) is used which has a key field definition that does not match the compile-time file's key field definition. A single-record input operation from the query file will perform server-to-workstation conversion on the key feedback values returned with the

operation. This may trigger invalid data errors if the returned key value formats do not match the expected formats from the compile-time file's key definition. This error situation can be avoided if:

- Only blocked record input operations are used, or
- A compile-time file with a key field definition which matches the query file used at program run time is selected.

Applications with Embedded Database File Overrides

If a VARPG application uses an OVRDBF (Override Database File) command to specify a different library or file name, the file open will fail if a file by the original name does not exist on the server.

The VARPG file open request goes through the iSeries DDM support, which attempts to locate and lock the file before applying the file override information. If this lock attempt fails on the original library/filename, then the open request fails. If the file lock succeeds, the request then passes to the iSeries database layer which applies the file override and performs the open on the redirected file name.

OS/400 File Data Conversions

Data is stored differently on the iSeries server than on the workstation. For this reason, VisualAge RPG data conversion occurs if your VisualAge RPG application calls OS/400 programs, accesses OS/400 data areas, or accesses OS/400 database files. This conversion ensures that data is represented correctly on both the workstation and on the iSeries server.

If a data structure is passed as a parameter to a call to an OS/400 program, or if the data structure represents an OS/400 data area, each subfield is converted based on its data type.

Note: In order for the data conversions to work, all database files and TO/FROM files must be externally described.

The following conversions are performed on VisualAge RPG data types:

Character Data:

Character data is converted from EBCDIC to ASCII and vice versa. All character data stored in the database must be tagged with an appropriate EBCDIC CCSID in order for the conversion to work correctly. Character data is converted depending on the CCSID of the character field and the code page of the ASCII workstation. If your application calls an OS/400 program or accesses an OS/400 data area, the CCSID of the job serving the call or the data area request is used when converting data.

For more information on tagging character data, see the the *DB2 Universal Database* section of the *Database and File Systems* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**.

Graphic data type:

The Graphic data type contains all DBCS characters without the SO and SI characters. Since this is the format that the system expects, the entire graphic data type can be used.

Note: If your application issues server I/O requests, you must tag the OS/400 Graphic fields with an appropriate DBCS CCSID in order for the VisualAge RPG conversion to work successfully.

Date, Time and Timestamp data types:
The conversion is the same as for character data types. Explicit CCSID tagging is not required for OS/400 database access.

Zoned Numeric data type:
Zoned numbers are converted so they can be displayed and printed on both the server and the workstation.

When negative EBCDIC zoned numbers are converted to ASCII, the sign portion of the last byte is converted to x'7'. See "Zoned-Decimal Format" on page 129.

Packed Decimal data type:
When EBCDIC packed numbers are converted to ASCII, the sign portion of the packed number is converted to x'C' if the number is positive and x'D' if the number is negative. See "Packed-Decimal Format" on page 126.

Binary data types:
Binary fields are reordered when this data is sent between the server and the workstation.

The following conversions are performed on database types:

Float data types:
Binary and Float fields are byte swapped when this data is sent between the server and the workstation. For example,
- A 2 byte integer field on the server containing '1 2' is converted to '2 1' on the workstation.
- A 2 byte integer field on the containing '2 1' on the workstation is converted to '1 2' on the server .
- A 4 byte integer field on the server containing '1 2 3 4' is converted to '4 3 2 1' on the workstaiton.
- A 4 byte integer field on the containing '4 3 2 1' on the workstation is converted to '1 2 3 4' on the server .

Hex Data Type and Character Data tagged with a CCSID of 65535
Data conversion does not occur if the character field is tagged with a CCSID of 65535 (implying no conversion should take place).

J, O, or E Data types:
The J (DBCS only), O (Mixed data) and E (Either all single byte or all double type) data types are treated as character fields in the VisualAge RPG program.

J, O, and E data types must be tagged with an appropriate CCSID. However, since they may contain DBCS characters they are special. On the server, DBCS characters for these data types are enclosed by the SO (Shift out) and SI (Shift In) characters. On the workstation, DBCS is not enclosed with the SO and SI characters.

When data is retrieved from the server, these characters are stripped and the character field is padded with two additional trailing blanks.

When data is sent to the server, you must ensure that there are enough trailing blanks in the character field so that they can be replaced by the appropriate number of SO and SI characters which must be added to the OS/400 DBCS field.

For example, assume that an O (mixed data) field is created in the database. On the workstation, this field contains the following before being written to the database.

```
sbDBsbDBblblblbl
Where sb = Single byte character.
      DB = Double byte character.
      bl = Single byte blank character.
```

In this example, this field is converted as follows:

```
sbSODBSIsbSODBSI
```

Note: The trailing single byte blanks are treated as insignificant and are replaced with the SO and SI characters appropriately.

Variable length fields:
For server I/O requests, the Binary portion is reordered when this data is sent between the server and the workstation. The character portion is converted based on the field CCSID.

OS/400 Database File Commitment Control

OS/400 commitment control allows you to process a group of database changes as a unit. This unit can be successfully applied to the database by issuing a commit operation. Changes associated with the unit can be rolled back if they cannot be successfully applied as a group.

The information you enter on the Define iSeries Information notebook allows you to define multiple OS/400 database files for your VisualAge RPG application. These files can exist on multiple servers. For more information on defining server information, see the online help and *Programming with VisualAge RPG*, SC09-2449-05.

A commitment control environment can only be started for one server. You can still use the database file on other servers. However those files cannot be opened under commitment control.

After the commitment control environment has been started for a server, you can open database files on that server using the COMMIT keyword on the file specification for the files you want opened under commitment control. Only files opened under commitment control are affected by subsequent COMMIT and ROLLBK operations. For more information on the COMMIT keyword, see "COMMIT{(rpg_name)}" on page 245.

After making the appropriate changes associated with your transaction, you can commit the changes to the database using the COMMIT operation code or you can rollback the database changes using the ROLLBK operation code.

Commitment control is ended when your VisualAge RPG application ends. If changes are pending in the database which have not been explicitly committed or rolled back, then an implicit rollback operation occurs at application termination.

You can write a program so that the decision to open a file under commitment control is made at run time. The COMMIT keyword on the file specification has a parameter which allows you to specify conditional commitment control. For more information on using the COMMIT keyword to control opening a file for commitment control at run time, see "COMMIT{(rpg_name)}" on page 245.

Level Checking: The VisualAge RPG supports level checking between a VisualAge RPG program and the database files being used.

The VisualAge RPG compiler provides the information required by level checking. Level checking occurs on a record-format basis when the file is opened unless you specify LVLCHK(*NO) when creating or changing the database file.

Note: If a level check occurs, it is handled as an I/O error.

Floating Point: Floating-point fields are not supported. If you process an externally-described OS/400 file with floating-point fields, the floating-point fields cannot be accessed by the VisualAge RPG application. When you create a new record, the floating-point fields in the record have a value of zero. When you update existing records, the floating-point fields are unchanged. You cannot use a floating-point field as a key field.

Locking Files: The OS/400 system allows a lock state (exclusive, exclusive allow read, shared for update, shared no update, or shared for read) to be placed on a file used during the execution of a job. Programs within a job are not affected by file lock states. A file lock state applies only when a program in another job tries to use the file concurrently. The file lock state can be allocated with the CL command ALCOBJ (Allocate Object). For more information on allocating resources and lock states, see the *CL and APIs* section of the *Programming* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**.

The OS/400 system places the following lock states on database files when it opens the files:
- Opened for INPUT: Lock state of Shared for read
- Opened for UPDATE: Lock state of Shared for update
- Opened for ADD: Lock state of Shared for update
- Opened for OUTPUT: Lock state of Shared for update.

Locking Records: When a record is read from a file that has been opened for update, a lock is applied to the record. Other programs on the server and other open instances in your VisualAge RPG application of the same file cannot read this record for update until the record lock is released.

You can read a record for input purposes even if the file is an update file by using the (N) operation code extender in the operation code field following the operation code name. The following operation codes cause a record to be locked if the operation code extender (N) is not specified:
- CHAIN
- READ
- READE
- READP
- READPE

The lock remains until one of the following occurs:
- The record is updated
- The record is deleted
- Another record is read from the file (for input or update)
- A SETLL or SETGT operation is performed against the file
- An UNLOCK operation is performed against the file
- An output operation defined by an output specification with no field names included is performed against the file

  Note: An output operation that adds a record to a file does not result in a record lock being released.

If your program attempts to read a record for update and the record is already locked by another file open instance, then the read operation waits until the record is unlocked. If the wait time exceeds the WAITRCD parameter on the file, an exception occurs. If your program does not handle this exception (RNQ1218), then the default error handler gets control. You have the option to retry the operation. This allows the program to continue as if the record lock timeout had not occurred.

Note: If the file has an INFSR subroutine specified when an I/O operation is performed on the file before the default error handler is given control, unexpected results can occur if the input operation that is retried is a sequential operation. This can occur if the file cursor has been modified.

Printer Files

The following rules apply to printer files:

- The record length for the printer device in a VisualAge RPG application must be less than or equal to the physical page width.
- An automatic form-feed is inserted when the value of the current line is equal to the value specified by the FORMLEN(n) keyword, where n is the number of lines per page, If FORMLEN is not specified, the default page length is 66 lines.
- If printing finishes in the middle of a printer file's page, a form-feed is added automatically.

The following restrictions apply to printer files:

- The PRINT feedback area (for number of pages and lines) is updated only after a POST operation.
- Major/minor return codes, I/O feedback areas, and open feedback areas are not supported.
- Overflow/fetch is not supported. However, when the line number reaches the value specified by FORMLEN (or the default of 66), a form feed takes place.
- Since overflow is not supported, the *OFL routine (for file exceptions and errors) is not supported. This means the *ROUTINE does not get updated with this value in the file feedback area.

Special Files

A special file allows you to specify an input and/output device that is not directly supported by the VisualAge RPG operations. The input and output operations for the file are controlled by a user-written routine. Use the PROCNAME keyword on the file specifications to define the special file handler.

Note: The user-written routine is a function contained in a Dynamic Link Library (DLL).

In Figure 25 on page 97, **fspecr** is a function within a C module which has been compiled and linked to a VisualAge RPG application. This example demonstrates how to perform local I/O on a workstation.

Note: In order for your VisualAge RPG program to connect with a DLL, you must create the DLL containing the C function definition referred to by the procedure keyword (PROCNAME) in the VisualAge RPG program. When you build your VisualAge RPG application, you must specify the list of libraries (LIB) and/or objects (OBJ) which contains all the functions that the application calls on the Build notebook.

```
F* ------------------------------------------------------------
F* ---------- ---   VRPG Special Files  --- ----------------------
F* ------------------------------------------------------------
F* Special file declaration. Explicit open and close.
F dino     cf   f   18        SPECIAL  USROPN PLIST(dinoplist)
F                                        PROCNAME('fspecr')
F                                        INFDS(info)
D* ------------------------------------------------------------
D* The INFDS- positions 38 to 42.
D info            DS
d  errinfo               38     42s 0
D* ------------------------------------------------------------
D* Used to display set indicators and read values.
D BoxId           M                   STYLE(*INFO)
D                                     BUTTON(*OK:*ENTER)
D                                     BUTTON(*ABORT:*IGNORE)
D* ------------------------------------------------------------
D* Input field associated with special file.
D fieldc          S             18            inz ('VRPGisGreat')
D* ------------------------------------------------------------
D* Extra, user-defined parameter for special file I/O operations.
D mySFparm        S             10            inz ('VRPGisGreat')
I* ------------------------------------------------------------
I* Input specification, i.e. fieldc is updated for the VRPG program
I* after each I/O operation performed on the special file dino.
I dino     NS
I                                  1   18  fieldc

I*
```

Figure 25. Using Special Files (Part 1 of 2)

```
I* ------------------------------------------------------------
C     *INZSR        BEGSR
C* ------------------------------------------------------------
C     dinoplist     PLIST
C                   PARM                    mySFparm
C                   EVAL                    fieldc = 'FIRSTINIT'
C     FIELDC        DSPLY     BoxId         Reply             9 0
C*-------------------------------------------------------------------
C                   OPEN      dino                                90
C   90'IND90'       DSPLY     BoxId         Reply
C   99'IND99'       DSPLY     BoxId         Reply
c*-------------------------------------------------------------------
C  N90              READ      dino                                9099
C*-------------------------------------------------------------------
C  N90FIELDC        DSPLY     BoxId         Reply
C   90'IND90'       DSPLY     BoxId         Reply
C   99'IND99'       DSPLY     BoxId         Reply
c*-------------------------------------------------------------------
C                   CLOSE     dino                                90
C*-------------------------------------------------------------------
C   90'IND90'       DSPLY     BoxId         Reply
C   99'IND99'       DSPLY     BoxId         Reply
C                   ENDSR
```

Figure 25. Using Special Files (Part 2 of 2)

```
#include  <stdlib.h>
#include  <memory.h>
#include  <stdio.h>
/* ------------------------------------------------------------,
|                   Special File Function                     |
'------------------------------------------------------------*/
 *
extern void fspecr ( char *option,       // VRPG provided
                     char *status,       // VRPG provided
                     char *error,        // VRPG provided
                     char *area,         // VRPG provided
                     char *mySFparm)     // User provided
{
/* ------------------------------------------------------------,
|                   Local Constants                           |
'------------------------------------------------------------*/
#define REC_SIZE 18             // Size of record to be read.
#define NORMAL_STATUS '0'       // Values for 'status' parameter.
#define ERROR_STATUS  '2'
#define EOF_STATUS    '1'
#define OPEN_ERROR    12345     // Update values for 'error' also
#define READ_ERROR    88888     // used for the *RECORD field in
#define OPTION_ERROR  99999     // in the INFDS.
 *
  static FILE *fp ;
 *
  int radix = 10 ;              // Required for the '_itoa' function.
 *
  int a     = 0 ;
  char temp[6];
  switch  (option[0]) {
    case 'O' :                        // Locally open the file.
      if ((fp=fopen("special.dat", "rb+")) == NULL) {
         a = OPEN_ERROR ;             // ASCII value of an open error
         _itoa( a, temp, radix ) ;    // is set here for the INFDS.
         memcpy( error, temp, 5 ) ;
         status[0] = ERROR_STATUS ; // Return status.
      }
      else {
         status[0] = NORMAL_STATUS ;
      }
 *
      break ;
 *
    case 'C' :                        // Local close ...
      fclose(fp) ;
      status[0] = NORMAL_STATUS ;
      break ;
```

Figure 26. C program for Special Files (Part 1 of 2)

```
 *
     case 'R' :                       // Local file open... read.
       fread(area, 1, REC_SIZE, fp) ;
       if (feof(fp)) {
         status[0]  = EOF_STATUS ;    // File read and EOF reached.
       }
       else if (ferror(fp)) {         // Check for any errors.
          a = READ_ERROR ;            // ASCII equivalent of a read error
          _itoa( a, temp, radix ) ;   // is set here for the INFDS.
          memcpy( error, temp, 5 ) ;
          status[0] = ERROR_STATUS ; // Return status.
       }
       else {
         status[0] = NORMAL_STATUS ;
       }
       break ;
 *
      default :
          a = OPTION_ERROR ;          // Set the ASCII equivalent of an
          _itoa( a, temp, radix ) ;   // option error for the INFDS.
          memcpy( error, temp, 5 ) ;
          status[0] = ERROR_STATUS ; // Return status.
          break ;
 *
      }
 *
      return ;
 *
#undef REC_SIZE
#undef NORMAL_STATUS
#undef ERROR_STATUS
#undef EOF_STATUS
#undef OPEN_ERROR
#undef READ_ERROR
#undef OPTION_ERROR
}
```

Figure 26. C program for Special Files (Part 2 of 2)

The example in Figure 27 illustrates how the OPEN and CLOSE operations are done implicitly by omitting the USROPN keyword from the File specifications.

```
F* ------------------------------------------------------------
F* ---------- ---   VRPG Special Files  --- ----------------------
F* ------------------------------------------------------------
F* Special file declaration. Implicit open and close.
F dino     cf   f   18        SPECIAL  PLIST(dinoplist)
F                                      PROCNAME('fspecr')
F                                      INFDS(info)
D* ------------------------------------------------------------
D* The INFDS- positions 38 to 42.
D info            DS
d  errinfo               38     42s 0
D* ------------------------------------------------------------
D* Used to display set indicators and field values.
D BoxId           M                   STYLE(*INFO)
D                                     BUTTON(*OK:*ENTER)
D                                     BUTTON(*ABORT:*IGNORE)
D* ------------------------------------------------------------
D* Input field associated with special file.
D fieldc          S             18            inz ('VRPGisGreat')
D* ------------------------------------------------------------
D* Extra, user-defined parameter for special file I/O operations.
D mySFparm        S             10            inz ('VRPGisGreat')
I* ------------------------------------------------------------
I* Input specification, i.e. fieldc is updated for the VRPG program
I* after each I/O operation performed on the special file dino.
I dino     NS
I                                  1   18  fieldc
I* ------------------------------------------------------------
C     *INZSR        BEGSR
C* ------------------------------------------------------------
C     dinoplist     PLIST
C                   PARM                    mySFparm
C                   EVAL                    fieldc = 'FIRSTINIT'
C     FIELDC        DSPLY     BoxId         Reply             9 0
C*------------------------------------------------------------------
C                   READ      dino                                9099
C*------------------------------------------------------------------
C  N90FIELDC        DSPLY     BoxId         Reply
C   90'IND90'       DSPLY     BoxId         Reply
C   99'IND99'       DSPLY     BoxId         Reply
C*------------------------------------------------------------------
C                   ENDSR
```

Figure 27. Opening and Closing Special Files Implicitly

Part 2. Data

This section provides information on using data in a program:

- Chapter 9, “Data Types and Data Formats,” on page 103 describes data type and formats
- “Literals” on page 149 describes literals
- Chapter 11, “Data Structures,” on page 157 describes data structures
- Chapter 12, “Using Arrays and Tables,” on page 171 describes Arrays and tables
- Chapter 13, “Editing Numeric Fields,” on page 191 describes how to edit numeric fields
- Chapter 14, “Initialization of Data,” on page 207 describes data initialization

Chapter 9. Data Types and Data Formats

This section describes the data types supported by VisualAge RPG and their special characteristics. The supported data types are:
- Basing Pointer
- Character
- Date
- Graphic
- Numeric
- Object
- Procedure Pointer
- Time
- Timestamp
- UCS-2

In addition, some of the data types allow different data formats. This section describes the difference between internal and external data formats, describes each format, and how to specify them.

Internal and External Formats

Numeric, date, and timestamp fields have an internal format that is independent of the external format. The **internal format** is the way the data is stored in the program. The **external format** is the way the data is stored in files.

You need to be aware of the internal format when:
- Passing parameters by reference
- Overlaying subfields in data structures

In addition, you may want to consider the internal format of numeric fields, when the runtime performance of arithmetic operations is important. For more information, see "Performance Considerations" on page 349.

There is a default internal and external format for numeric and date-time data types. You can specify an internal format for a specific field on a definition specification. Similarly, you can specify an external format for a program-described field on the corresponding input or output specification.

For fields in an externally-described file, the external data format is specified in the data description specifications in position 35. You cannot change the external format of externally-described fields, with one exception. If you specify EXTBININT on a control specification, any binary field with zero decimal positions will be treated as having an integer external format.

For subfields in externally-described data structures, the data formats specified in the external description are used as the internal formats of the subfields by the compiler.

Internal Format

The default internal format for numeric standalone fields is packed-decimal. The default internal format for numeric data structure subfields is zoned-decimal. To

specify a different internal format, specify the format desired in position 40 on the definition specification for the field or subfield.

The default format for date, time, and timestamp fields is *ISO. In general, it is recommended that you use the default ISO internal format, especially if you have a mixture of external format types.

For date, time, and timestamp fields, you can use the DATFMT and TIMFMT keywords on the control specification to change the default internal format, if desired, for *all* date-time fields in the program. You can use the DATFMT or TIMFMT keyword on a definition specification to override the default internal format of an *individual* date-time field.

External Format

If you have numeric, character, or date-time fields in program-described files, you can specify their external format. Valid external numeric formats are: binary, integer, packed-decimal, zoned-decimal, unsigned or float. The external format does not affect the way in which a field is processed. However, you may be able to improve performance of arithmetic operations, depending on the internal format specified. For more information, see "Performance Considerations" on page 349.

Specifying an External Format for a Numeric Field

The following table shows how to specify the external format of numeric program-described fields. For more information on each format type, see the appropriate section in the remainder of this section.

Table 13. Entries and Locations for Specifying External Formats

Type of Field	Specification	Using
Input	Input	Position 36
Output	Output	Position 52
Array or Table	Definition	EXTFMT keyword

For any of these fields in Table 13, specify one of the following valid external numeric formats:

B Binary
F Float
I Integer
L Left sign
P Packed decimal
R Right sign
S Zoned decimal
U Unsigned

The default external format for float numeric data is called the external display representation. The format for 4-byte float data is:

```
+n.nnnnnnnE+ee, where + represents the sign (+ or -)
                      n represents digits in the mantissa
                      e represents digits in the exponent
```

The format for 8-byte float data is:

```
+n.nnnnnnnnnnnnnnnE+eee
```

Note that a 4-byte float value occupies 14 positions and an 8-byte float value occupies 23 positions.

For numeric data other than float, the default external format is zoned decimal. The external format for compile-time arrays and tables must be zoned-decimal, left-sign or right-sign.

For float compile-time arrays and tables, the compile-time data is specified as either a numeric literal or a float literal. Each element of a 4-byte float array requires 14 positions in the source record; each element of an 8-byte float array requires 23 positions.

Non-float numeric fields defined on input specifications, calculation specifications, or output specifications with no corresponding definition on a definition specification are stored internally in packed-decimal format.

Specifying an External Format for a Character, Graphic, or UCS-2 Field

For any of the input and output fields in Table 13 on page 104, specify one of the following valid external data formats:

A Character (valid for character and indicator data)

N Indicator (valid for character and indicator data)

G Graphic (valid for graphic data)

C UCS-2 (valid for UCS-2 data)

The EXTFMT keyword can be used to specify the data for an array or table in UCS-2 format.

Specify the *VAR data attribute in positions 31-34 on an input specification and in positions 53-80 on an output specification for variable-length character, graphic, or UCS-2 data.

Specifying an External Format for a Date-Time Field

If you have date, time, and timestamp fields in program-described files, then you *must* specify their external format. You can specify a default external format for all date, time, and timestamp fields in a program-described file by using the DATFMT and TIMFMT keywords on a File-Description specification. You can specify an external format for a particular field as well. Specify the desired format in positions 31-34 on an Input specification. Specify the appropriate keyword and format in positions 53-80 on an Output specification.

For more information on each format type, see the appropriate section in the remainder of this chapter.

Basing Pointer Data Type

Basing pointers are used to locate the storage for based variables. The storage is accessed by defining a field, array, or data structure as based on a particular basing pointer variable and setting the basing pointer variable to point to the required storage location.

For example, consider the based variable MY_FIELD, a character field of length 5, which is based on the pointer PTR1. The based variable does not have a fixed location in storage. You must use a pointer to indicate the current location of the storage for the variable.

Suppose that the following is the layout of some area of storage:

```
-------------------------------------------------------
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
-------------------------------------------------------
```

If we set pointer PTR1 to point to the G,

```
   PTR1-------------------.
                          |
                          V
-------------------------------------------------------
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
-------------------------------------------------------
```

MY_FIELD is now located in storage starting at the 'G', so its value is 'GHIJK'. If the pointer is moved to point to the 'J', the value of MY_FIELD becomes 'JKLMN':

```
               PTR1-------------------.
                                      |
                                      V
-------------------------------------------------------
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
-------------------------------------------------------
```

If MY_FIELD is now changed by an EVAL statement to 'HELLO', the storage starting at the 'J' would change:

```
               PTR1-------------------.
                                      |
                                      V
-------------------------------------------------------
| A | B | C | D | E | F | G | H | I | H | E | L | L | O | O |
-------------------------------------------------------
```

Use the BASED keyword on the definition specification (see "BASED(basing_pointer_name)" on page 266) to define a basing pointer for a field. Basing pointers have the same scope as the based field.

The length of the basing pointer field must be 4 bytes long and must be aligned on a 4 byte boundary. This requirement for boundary alignment can cause a pointer subfield of a data structure not to follow the preceding field directly, and can cause multiple occurrence data structures to have non-contiguous occurrences. For more information on the alignment of subfields, see "Aligning Data Structure Subfields" on page 160.

The default initialization value for basing pointers is *NULL.

Notes:

1. When coding basing pointers, you must be sure that you set the pointer to storage that is large enough and of the correct type for the based field. Figure 31 on page 109 shows some examples of how *not* to code basing pointers.
2. You can add or subtract an offset from a pointer in an expression, for example EVAL ptr = ptr + offset. When doing pointer arithmetic be aware that it is your responsibility to ensure that you are still pointing within the storage of the item you are pointing to. In most cases no exception will be issued if you point before or after the item.

   When subtracting two pointers to determine the offset between them, the pointers must be pointing to the same space, or the same type of storage. For example, you can subtract two pointers in static storage, or two pointers in automatic storage, or two pointers within the same user space.

Setting a Basing Pointer

You set or change the location of the based variable by setting or changing the basing pointer in one of the following ways:

- Initializing with INZ(%ADDR(FLD)) where FLD is a non-based variable
- Assigning the pointer to the result of %ADDR(X) where X is any variable
- Assigning the pointer to the value of another pointer
- Using ALLOC or REALLOC (See "ALLOC (Allocate Storage)" on page 505 and "REALLOC (Reallocate Storage with New Length)" on page 666 for examples.)
- Moving the pointer forward or backward in storage using pointer arithmetic:

```
EVAL        PTR = PTR + offset
```

("offset" is the distance in bytes that the pointer is moved)

Examples

Figure 28 shows how to define a based data structure.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *
 *  Define a based data structure, array and field.
 *  If PTR1 is not defined, it will be implicitly defined
 *  by the compiler.
 *
 *  Note that before these based fields or structures can be used,
 *  the basing pointer must be set to point to the correct storage
 *  location. PTR1 will be set to a valid storage address before the
 *  DSbased data structure is used.
 *
D DSbased         DS                  BASED(PTR1)
D    Field1                   1  16A
D     Field2                     2
D
D ARRAY           S             20A   DIM(12) BASED(PTR2)
D
D Temp_fld        S               *   BASED(PTR3)
D
D PTR2                            *   INZ
D PTR3                            *   INZ(*NULL)
```

Figure 28. Defining Based Structures and Fields

The following shows how you can add and subtract offsets from pointers and also determine the difference in offsets between two pointers.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
D P1              s               *
D P2              s               *
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01+++++++++++++Opcode(E)+Extended Factor 2++++++++++++++++++++++++++++
 *
 * Allocate 20 bytes of storage for pointer P1.
C                   ALLOC     20            P1
 * Initialize the storage to 'abcdefghij'
C                   EVAL      %STR(P1:20) = 'abcdefghij'
 * Set P2 to point to the 9th byte of this storage.
C                   EVAL      P2 = P1 + 8
 * Show that P2 is pointing at 'i'.  %STR returns the data that
 * the pointer is pointing to up to but not incuding the first
 * null-terminator x'00' that it finds, but it only searches for
 * the given length, which is 1 in this case.
C                   EVAL      Result = %STR(P2:1)
C                   DSPLY                   Result            1
 * Set P2 to point to the previous byte
C                   EVAL      P2 = P2 - 1
 * Show that P2 is pointing at 'h'
C                   EVAL      Result = %STR(P2:1)
C                   DSPLY                   Result
 * Find out how far P1 and P2 are apart.  (7 bytes)
C                   EVAL      Diff = P2 - P1
C                   DSPLY                   Diff              5 0
 * Free P1's storage
C                   DEALLOC                 P1
C                   RETURN
```

Figure 29. Pointer Arithmetic

Figure 30 shows how to obtain the number of days in Julian format, if the Julian date is required.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7....+....
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DATFMT(*JUL)
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
D JulDate         S               D   INZ(D'95/177')
D                                     DATFMT(*JUL)
D JulDS           DS                  BASED(JulPTR)
D  Jul_yy                        2  0
D  Jul_sep                       1
D  Jul_ddd                       3  0
D JulDay          S              3  0
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01+++++++++++++++Opcode(E)+Extended Factor 2++++++++++++++++++++++++++++
 *
 * Set the basing pointer for the structure overlaying the
 * Julian date.
C                   EVAL      JulPTR = %ADDR(JulDate)
 * Extract the day portion of the Julian date
C                   EVAL      JulDay = Jul_ddd
```

Figure 30. Obtaining a Julian Date

When coding basing pointers, make sure that the pointer is set to storage that is large enough and of the correct type for the based field. Figure 31 shows some examples of how *not* to code basing pointers.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
  *
D chr10           S             10a   based(ptr1)
D char100         S            100a   based(ptr1)
D p1              S              5p 0 based(ptr1)
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01+++++++++++++++Opcode(E)+Extended Factor 2++++++++++++++++++++++++++++
  *
  *
  * Set ptr1 to the address of p1, a numeric field
  * Set chr10 (which is based on ptr1) to 'abc'
  * The data written to p1 will be unreliable because of the data
  * type incompatibility.
  *
C                   EVAL      ptr1 = %addr(p1)
C                   EVAL      chr10 = 'abc'
  *
  * Set ptr1 to the address of chr10, a 10-byte field.
  * Set chr100, a 100-byte field, all to 'x'
  * 10 bytes are written to chr10, and 90 bytes are written in other
  * storage, the location being unknown.
  *
C                   EVAL      ptr1 = %addr(chr10)
C                   EVAL      chr100 = *all'x'
```

Figure 31. How Not to Code Basing Pointers

Character Data Type

The character data type represents character values and may have one of the following formats:

A Character

N Indicator

G Graphic

C UCS-2

Character data may contain one or more single-byte or double-byte characters, depending on the format specified. Character, graphic, and UCS-2 fields can also have either a fixed or variable-length format. Operation codes which operate on strings accept character data. The following table summarizes the different character data-type formats.

Character Data Type	Number of Bytes	CCSID
Character	One or more **single-byte** characters that are fixed or variable in length	assumed to be the CCSID of the workstation
Indicator	One **single-byte** character that is fixed in length	assumed to be the CCSID of the workstation
Graphic	One or more **double-byte** characters that are fixed or variable in length	CCSID of the workstation or a valid user-defined double-byte CCSID
UCS-2	One or more **double-byte** characters that are fixed or variable in length	13488 (UCS-2 version 2.0)

For information on the CCSIDs of character data, see "Conversion between Character, Graphic and UCS-2 Data" on page 119.

The default initialization value for non-indicator character fields is blanks.

Indicators are a special type of character data. Indicator data consists of the indicator and the field specified with the COMMIT keyword on the file description specification. Indicators are all one byte long and can only contain the character values '0' and '1'. The default value of indicators is '0'.

Character Format

The fixed-length character format is one or more bytes long with a set length.

For information on the variable-length character format, see "Variable-Length Character, Graphic, and UCS-2 Format" on page 113.

You define a character field by specifying A in the Data-Type entry of the appropriate specification. You can also define one using the LIKE keyword on the definition specification where the parameter is a character field.

The default initialization value is blanks.

Indicator Format

The indicator format is a special type of character data. Indicators are all one byte long and can only contain the character values '0' (off) and '1' (on). They are generally used to indicate the result of an operation or to condition (control) the processing of an operation. The default value of indicators is '0'.

You define an indicator field by specifying N in the Data-Type entry of the appropriate specification. You can also define an indicator field using the LIKE keyword on the definition specification where the parameter is an indicator field. Indicator fields are also defined implicitly with the COMMIT keyword on the file description specification.

The rules for defining indicator variables are:

- Indicators can be defined as standalone fields, subfields, prototyped parameters, and procedure return values.
- If an indicator variable is defined as a prerun-time or compile-time array or table, the initialization data must consist of only '0's and '1's.

  Note: If an indicator contains a value other than '0' or '1' at runtime, the results are unpredictable.
- If the keyword INZ is specified, the value must be one of '0', *OFF, '1', or *ON.
- The keyword VARYING cannot be specified for an indicator field.

The rules for using indicator variables are:

- The default initialization value for indicator fields is '0'.
- Operation code CLEAR sets an indicator variable to '0'.
- Blank-after function applied to an indicator variable sets it to '0'.
- If an array of indicators is specified as the result of a MOVEA(P) operation, the padding character is '0'.
- Indicators may be used as search arguments where the external key is a character of length 1.

Graphic Format

The graphic format is a character string where each character is represented by 2 bytes.

The difference between single- byte character and double-byte graphic data is shown in the following figure:

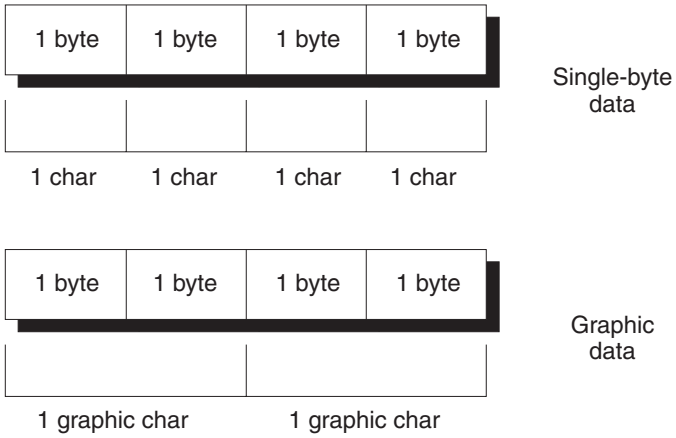


Figure 32. Comparing Single-byte and Graphic Data

The length of a graphic field, in bytes, is two times the number of graphic characters in the field.

If a record is added, the database file and graphic fields are not specified for output, double-byte blanks are placed in the fields for output. Blanks are placed in output fields in the following conditions:
- The fields are not specified for output on the output specification.
- Conditioning indicators are not satisfied for the field.

Graphic data may be fixed or variable length. The fixed-length graphic format is a character string with a set length where each character is represented by 2 bytes.

For information on the variable-length character format, see "Variable-Length Character, Graphic, and UCS-2 Format" on page 113.

You define a graphic field by specifying G in the Data-Type entry of the appropriate specification. You can also define one using the LIKE keyword on the definition specification where the parameter is a graphic field.

The default initialization value for graphic data is the double byte blank. Its hexadecimal value depends on the code page installed on your workstation. The value of *HIVAL is X'FFFF', and the value of *LOVAL is X'0000'.

UCS-2 Format

The Universal Character Set (UCS-2) format is a character string where each character is represented by 2 bytes. This character set can encode the characters for many written languages.

The length of a UCS-2 field, in bytes, is two times the number of UCS-2 characters in the field.

The fixed-length UCS-2 format is a character string with a set length where each character is represented by 2 bytes.

For information on the variable-length UCS-2 format, see "Variable-Length Character, Graphic, and UCS-2 Format."

You define a UCS-2 field by specifying C in the Data-Type entry of the appropriate specification. You can also define one using the LIKE keyword on the definition specification where the parameter is a UCS-2 field.

The default initialization value for UCS-2 data is X'0020'. The value of *HIVAL is X'FFFF', *LOVAL is X'0000', and the value of *BLANKS is X'0020'.

For more information on the UCS-2 format, see the *CL and APIs* section of the *Programming* category in the **Information Center** at this Web site - **http://www.ibm.com/eserver/iseries/infocenter**.

Variable-Length Character, Graphic, and UCS-2 Format

Variable-length character fields have a declared maximum length and a current length that can vary while a program is running. The length is measured in single bytes for the character format and in double bytes for the graphic and UCS-2 formats. The storage allocated for variable-length character fields is 2 bytes longer than the declared maximum length. The leftmost 2 bytes are an unsigned integer field containing the current length in characters, graphic characters, or UCS-2 characters. The actual character data starts at the third byte of the variable-length field.

Figure 33 shows how variable-length character fields are stored:

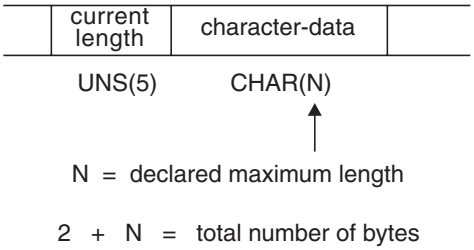


Figure 33. Character Fields with Variable-Length Format

Figure 34 shows how variable-length graphic fields are stored. UCS-2 fields are stored similarly.

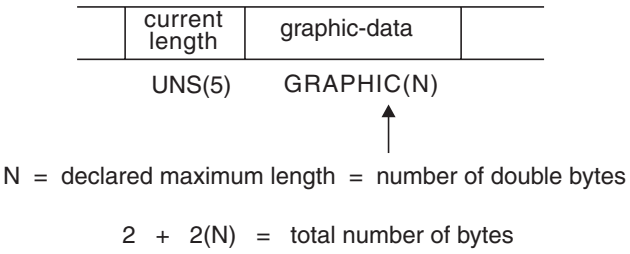


Figure 34. Graphic Fields with Variable-Length Format

Note: Only the data up to and including the current length is significant.

You define a variable-length character field by specifying A (character), G (graphic), or C (UCS-2) and the keyword VARYING on a definition specification. It can also be defined using the LIKE keyword on a definition specification where the parameter is a variable-length character field.

You can refer to external variable-length fields, on an input or output specification, with the *VAR data attribute.

The default initialization value is the null string (''); a value with length zero.

For examples of using variable-length fields, see:
- "Using Variable-Length Fields" on page 117
- "%LEN (Get or Set Length)" on page 452
- "%CHAR (Convert to Character Data)" on page 416
- "%REPLACE (Replace Character String)" on page 466

The variable-length format is also available for graphic data.

Rules for Variable-Length Character, Graphic, and UCS-2 Formats

The following rules apply when defining variable-length fields:

- The declared length of the field can be from 1 to 65535 single-byte characters and from 1 to 16383 double-byte graphic or UCS-2 characters.
- The current length may be any value from 0 to the maximum declared length for the field.
- The field may be initialized using keyword INZ. The initial value is the exact value specified and the initial length of the field is the length of the initial value. The field is padded with blanks for initialization, but the blanks are not included in the length.
- In all cases except subfields defined using positional notation, the length entry (positions 33-39 on the definition specifications) contains the maximum length of the field not including the 2-byte length.
- For subfields defined using positional notation, the length includes the 2-byte length. As a result, a variable-length subfield may be 65537 single bytes long or 16384 double bytes long for an unnamed data structure.
- The keyword VARYING cannot be specified for a data structure.
- For variable-length prerun-time arrays, the initialization data in the file is stored in variable format, including the 2-byte length prefix.
- Since prerun-time array data is read from a file and files have a maximum record length of 32766, variable-length prerun-time arrays have a maximum size of 32764 single-byte characters, or 16382 double-byte graphic or UCS-2 characters.
- A variable-length array or table may be defined with compile-time data. The trailing blanks in the field of data are not significant. The length of the data is the position of the last non-blank character in the field. This is different from prerun-time initialization since the length prefix cannot be stored in compile-time data.
- For graphic compile time data, single-byte blanks are considered to be significant data. Compile time data for graphic arrays and tables must be padded with double-byte blanks. Single-byte blanks are considered non blanks.
- *LIKE DEFINE cannot be used to define a field like a variable-length field.

The following is an example of defining variable-length character fields:

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++++

 * Standalone fields:
D var5            S              5A   VARYING
D var10           S             10A   VARYING INZ('0123456789')
D max_len_a       S          32767A   VARYING

 * Prerun-time array:
D arr1            S            100A   VARYING FROMFILE(dataf)

 * Data structure subfields:
D ds1             DS

 *   Subfield defined with length notation:
D   sf1_5                       5A   VARYING
D   sf2_10                     10A   VARYING INZ('0123456789')

 *   Subfield defined using positional notation: A(5)VAR
D   sf4_5               101    107A   VARYING


 *   Subfields showing internal representation of varying:
D   sf7_25                   100A   VARYING
D   sf7_len                    5I 0 OVERLAY(sf7_25:1)
D   sf7_data                 100A   OVERLAY(sf7_25:3)

 * Procedure prototype
D Replace         PR         32765A   VARYING
D   String                   32765A   CONSTANT VARYING OPTIONS(*VARSIZE)
D   FromStr                  32765A   CONSTANT VARYING OPTIONS(*VARSIZE)
D   ToStr                    32765A   CONSTANT VARYING OPTIONS(*VARSIZE)
D   StartPos                     5U 0 VALUE
D   Replaced                     5U 0 OPTIONS(*OMIT)
```

Figure 35. Defining Variable-Length Character and UCS-2 Fields

The following is an example of defining variable-length graphic and UCS-2 fields:

```
* .. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+...
DName++++++++++ETDsFrom+++To/L+++IDc.Functions+++++++++++++++++
 *---------------------------------------------------------------
 * Graphic fields
 *---------------------------------------------------------------
 * Standalone fields:
D GRA20           S             20G   VARYING
D MAX_LEN_G       S          16383G   VARYING
 * Prerun-time array:
D ARR1            S            100G   VARYING FROMFILE(DATAF)
 * Data structure subfields:
D DS1             DS
 *   Subfield defined with length notation:
D   SF3_20                      20G   VARYING
 *   Subfield defined using positional notation: G(10)VAR
D   SF6_10               11     32G   VARYING
 *---------------------------------------------------------------
 *   UCS-2 fields
 *---------------------------------------------------------------
D MAX_LEN_C       S          16383C   VARYING
D FLD1            S              5C   INZ(%UCS2('ABCDE')) VARYING
D FLD2            S              2C   INZ(U'01230123') VARYING
D FLD3            S              2C   INZ(*HIVAL) VARYING
D DS_C            DS
D   SF3_20_C                    20C   VARYING
 *   Subfield defined using positional notation: C(10)VAR
D   SF_110_C             11     32C   VARYING
```

Figure 36. Defining Variable-Length Graphic and UCS-2 Fields

Using Variable-Length Fields

The length part of a variable-length field represents the current length of the field measured in characters. For character fields, this length also represents the current length in bytes. For double-byte fields (graphic and UCS-2), this represents the length of the field in double bytes. For example, a UCS-2 field with a current length of 3 is 3 double-byte characters long, and 6 bytes long.

The following sections describe how to best use variable-length fields and how the current length changes when using different operation codes.

How the Length of the Field is Set: When a variable-length field is initialized using INZ, the initial length is set to be the length of the initialization value. For example, if a character field of length 10 is initialized to the value 'ABC', the initial length is set to 3.

The EVAL operation changes the length of a variable-length target. For example, if a character field of length 10 is assigned the value 'XY', the length is set to 2.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D fld                       10A        VARYING
 * It does not matter what length 'fld' has before the
 * EVAL; after the EVAL, the length will be 2.
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                   EVAL      fld = 'XY'
```

The CLEAR operation changes the length of a variable-length field to 0.

The PARM operation sets the length of the result field to the length of the field in Factor 2, if specified.

Fixed form operations MOVE, MOVEL, CAT, SUBST and XLATE do not change the length of variable-length result fields. For example, if the value 'XYZ' is moved using MOVE to a variable-length character field of length 10 whose current length is 2, the length of the field will not change and the data will be truncated.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++
D fld                     10A        VARYING
      * Assume fld has a length of 2 before the MOVEL.
      * After the first MOVEL, it will have a value of 'XY'
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                   MOVEL     'XYZ'      fld
      * After the second MOVEL, it will have the value '1Y'
C                   MOVEL     '1'        fld
```

Note: The recommended use for MOVE and MOVEL, as opposed to EVAL, is for changing the value of fields that you want to be temporarily fixed in length. An example is building a report with columns whose size may vary from day to day, but whose size should be fixed for any given run of the program.

When a field is read from a file (Input specifications), the length of a variable-length field is set to the length of the input data.

The ″Blank After″ function of Output specifications sets the length of a variable-length field to 0.

You can set the length of a variable-length field yourself using the %LEN builtin function on the left-hand-side of an EVAL operation.

How the Length of the Field is Used: When a variable-length field is used for its value, its current length is used. For the following example, assume 'result' is a fixed length field with a length of 7.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++
D fld                     10A        VARYING
 * For the following EVAL operation
 *    Value of 'fld'    Length of 'fld'   'result'
 *    --------------    ---------------   -----------
 *    'ABC'             3                 'ABCxxx '
 *    'A'               1                 'Axxx   '
 *    ''                0                 'xxx    '
 *    'ABCDEFGHIJ'      10                'ABCDEFG'
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                   EVAL      result = fld + 'xxx'
 * For the following MOVE operation, assume 'result'
 * has the value '.......' before the MOVE.
 *    Value of 'fld'    Length of 'fld'   'result'
 *    --------------    ---------------   -----------
 *    'ABC'             3                 '....ABC'
 *    'A'               1                 '......A'
 *    ''                0                 '.......'
 *    'ABCDEFGHIJ'      10                'DEFGHIJ'
C                   MOVE      fld           result
```

Why You Should Use Variable-Length Fields: Using variable-length fields for temporary variables can improve the performance of string operations, as well as

making your code easier to read since you do not have to save the current length of the field in another variable for %SUBST, or use %TRIM to ignore the extra blanks.

If a subprocedure is meant to handle string data of different lengths, using variable-length fields for parameters and return values of prototyped procedures can enhance both the performance and readability of your calls and your procedures. You will not need to pass any length parameters within your subrocedure to get the actual length of the parameter.

Conversion between Character, Graphic and UCS-2 Data

Note: If graphic CCSIDs are ignored (CCSID(*GRAPH:*IGNORE) was specified on the control specification or CCSID(*GRAPH) was not specified at all), graphic data is not considered to have a CCSID and conversions are not supported between graphic data and UCS-2 data.

Character, graphic, and UCS-2 data can have different CCSIDs (Coded Character Set IDs). Conversion between these data types depends on the CCSID of the data.

CCSIDs of Data

The CCSID of character data is only considered when converting between character and UCS-2 data.

When converting between character and UCS-2 or graphic data, the CCSID of the character data is assumed to be the CCSID of the workstation.

The CCSID of UCS-2 data defaults to 13488. This default can be changed using the CCSID(*UCS2) keyword on the Control specification. The CCSID for program-described UCS-2 fields can be specified using the CCSID keyword on the Definition specification. The CCSID for externally-described UCS-2 fields comes from the external file.

Note: UCS-2 fields are defined in DDS by specifying a data type of G and a CCSID of 13488.

The CCSID of graphic data defaults to the value specified in the CCSID(*GRAPH) keyword on the Control specification. The CCSID for program-described graphic fields can be specified using the CCSID keyword on the Definition specification. The CCSID for externally-described graphic fields comes from the external file.

Date Data

Date fields have a predetermined size and format. They can be defined on the definition specification. Leading and trailing zeros are required for all date data.

Date constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Dates used for I/O operations such as input fields, output fields or key fields are converted (if required) to the necessary format for the operation.

The default internal format for date variables is *ISO. This default internal format can be overridden globally by the control specification keyword DATFMT and individually by the definition specification keyword DATFMT.

The hierarchy used when determining the internal date format and separator for a date field is:
1. From the DATFMT keyword specified on the definition specification
2. From the DATFMT keyword specified on the control specification
3. *ISO

There are three kinds of date data formats, depending on the range of years that can be represented. This leads to the possibility of a date overflow or underflow condition occurring when the result of an operation is a date outside the valid range for the target field. The formats and ranges are as follows:

Number of Digits in Year	Range of Years
2 (*YMD, *DMY, *MDY, *JUL)	1940 to 2039
3 (*CYMD, *CDMY, *CMDY)	1900 to 2899
4 (*ISO, *USA, *EUR, *JIS, *LONGJUL)	0001 to 9999

Table 14 lists the formats for date data and their separators:

For examples on how to code date fields, see the examples in:
- "Date Operations" on page 359
- "Moving Date-Time Data" on page 370
- "ADDDUR (Add Duration)" on page 502
- "MOVE (Move)" on page 604
- "EXTRCT (Extract Date/Time/Timestamp)" on page 579
- "SUBDUR (Subtract Duration)" on page 693
- "TEST (Test Date/Time/Timestamp)" on page 700

Table 14. RPG-defined date formats and separators for Date data type

Format Name	Description	Format (Default Separator)	Valid Separators	Length	Example
2-Digit Year Formats					
*MDY	Month/Day/Year	mm/dd/yy	/ - . , '&'	8	01/15/96
*DMY	Day/Month/Year	dd/mm/yy	/ - . , '&'	8	15/01/96
*YMD	Year/Month/Day	yy/mm/dd	/ - . , '&'	8	96/01/15
*JUL	Julian	yy/ddd	/ - . , '&'	6	96/015
4-Digit Year Formats					
*ISO	International Standards Organization	yyyy-mm-dd	-	10	1996-01-15
*USA	IBM® USA Standard	mm/dd/yyyy	/	10	01/15/1996
*EUR	IBM European Standard	dd.mm.yyyy	.	10	15.01.1996
*JIS	Japanese Industrial Standard Christian Era	yyyy-mm-dd	-	10	1996-01-15

The following table lists the *LOVAL, *HIVAL, and default values for all the date formats:

Table 15. Date Values

Format name	Description	*LOVAL	*HIVAL	Default Value
2-Digit Year Formats				
*MDY	Month/Day/Year	01/01/40	12/31/39	01/01/40
*DMY	Day/Month/Year	01/01/40	31/12/39	01/01/40
*YMD	Year/Month/Day	40/01/01	39/12/31	40/01/01
*JUL	Julian	40/001	39/365	40/001
4-Digit Year Formats				
*ISO	International Standards Organization	0001-01-01	9999-12-31	0001-01-01
*USA	IBM USA Standard	01/01/0001	12/31/9999	01/01/0001
*EUR	IBM European Standard	01.01.0001	31.12.9999	01.01.0001
*JIS	Japanese Industrial Standard Christian Era	0001-01-01	9999-12-31	0001-01-01

Separators

When coding a date format on a MOVE, MOVEL or TEST operation, separators are optional for character fields. To indicate that there are no separators, specify the format followed by a zero. For more information on how to code date formats without separators see “MOVE (Move)” on page 604, “MOVEL (Move Left)” on page 626 and “TEST (Test Date/Time/Timestamp)” on page 700.

Formats for MOVE, MOVEL, and TEST Operations

Several formats are also supported for fields used by the MOVE, MOVEL, and TEST operations only. This support is provided for compatibility with externally defined values that are already in a 3-digit year format and the 4-digit year *LONGJUL format.

Table 16 lists the valid externally defined date formats that can be used in Factor 1 of a MOVE, MOVEL, and TEST operation.

Table 16. Externally defined date formats and separators

Format Name	Description	Format (Default Separator)	Valid Separators	Length	Example (April 25, 2001)
3-Digit Year Formats(1.)					
*CYMD	Century Year/Month/Day	cyy/mm/dd	/ - . , ’&’	9	101/04/25
*CMDY	Century Month/Day/Year	cmm/dd/yy	/ - . , ’&’	9	104/25/01
*CDMY	Century Day/Month/Year	cdd/mm/yy	/ - . , ’&’	9	125/04/01
4-Digit Year Formats					
*LONGJUL	Long Julian	yyyy/ddd	/ - . , ’&’	8	2001/115

Table 16. Externally defined date formats and separators (continued)

Format Name	Description	Format (Default Separator)	Valid Separators	Length	Example (April 25, 2001)
Notes: 1. Valid values for the century character 'c' are: 'c' Years 0 1900-1999 1 2000-2099 . . . . . . 9 2800-2899					

Separators are optional for character fields in the *CYMD format. To indicate that there are no separators you can specify *CYMD0.

Numeric Data Type

Numeric data consists of any data defined as having zero or more decimal positions. Numeric data has one of the following formats:

- "Binary Format"
- "Float Format" on page 124
- "Integer Format" on page 126
- "Packed-Decimal Format" on page 126
- "Unsigned Format" on page 128
- "Zoned-Decimal Format" on page 129

The default initialization value for numeric fields is zeroes.

Binary Format

Binary format means that the sign (positive or negative) is in the leftmost bit of the field and the integer value is in the remaining bits of the field. Positive numbers have a zero in the sign bit; negative numbers have a one in the sign bit and are in twos complement form. In binary format, each field must be either 2 or 4 bytes long.

A binary field can be from one to nine digits in length and can be defined with decimal positions. If the length of the field is from one to four digits, the compiler assumes a binary field length of 2 bytes. If the length of the field is from five to nine digits, the compiler assumes a binary field length of 4 bytes.

Program-Described File

Every input field read in binary format is assigned a field length (number of digits) by the compiler. A length of 4 is assigned to a 2-byte binary field; a length of 9 is assigned to a 4-byte binary field, if the field is not defined elsewhere in the program. Because of these length restrictions, the highest decimal value that can be assigned to a 2-byte binary field is 9999 and the highest decimal value that can be assigned to a 4-byte binary field is 999 999 999. In general, a binary field of n digits can have a maximum value of n 9s. This discussion assumes zero decimal positions.

For program-described files, specify binary input, binary output, and binary array or table fields with the following entries:

- Binary input field: Specify B in position 36 of the input specifications.
- Binary output field: Specify B in position 52 of the output specifications. This position must be blank if editing is specified.

  The length of a field to be written in binary format cannot exceed nine digits. If the length of the field is from one to four digits, the compiler assumes a binary field length of 2 bytes. If the length of the field is from five to nine digits, the compiler assumes a binary field length of 4 bytes.

  Because a 2-byte field in binary format is converted by the compiler to a decimal field with 1 to 4 digits, the input value may be too large. If it is, the leftmost digit of the number is dropped. For example, if a four digit binary input field has a binary value of hexadecimal 6000, the compiler converts this to 24 576 in decimal. The 2 is dropped and the result is 4576. Similarly, the input value may be too large for a 4-byte field in binary format. If the binary fields have zero (0) decimal positions, then you can avoid this conversion problem by defining integer fields instead of binary fields.

  Note: Binary input fields cannot be defined as match or control fields.
- Binary array or table field: Specify B in position 40 of the definition specifications. The external format for compile-time arrays and tables must not be binary.

Externally Described File

For an externally-described file, the data format is specified in position 35 of the data description specifications. The number of digits in the field is exactly the same as the length in the DDS description. For example, if you define a binary field in your DDS specification as having 7 digits and 0 decimal positions, the data is handled as follows:

1. The field is defined as a 4-byte binary field in the input specification
2. A Packed(7,0) field is generated for the field in the VisualAge RPGprogram.

If you want to retain the complete binary field information, redefine the field as a binary subfield in a data structure or as a binary standalone field. Note that an externally-described binary field may have a value outside of the range allowed by VARPG binary fields. If the externally-described binary field has zero (0) decimal positions then you can avoid this problem. To do so, you define the externally-described binary field on a definition specification and specify the EXTBININT keyword on the control specification. This will change the external format of the externally-described field to that of a signed integer.

Figure 37 on page 124 shows what the decimal number 8191 looks like in various formats.

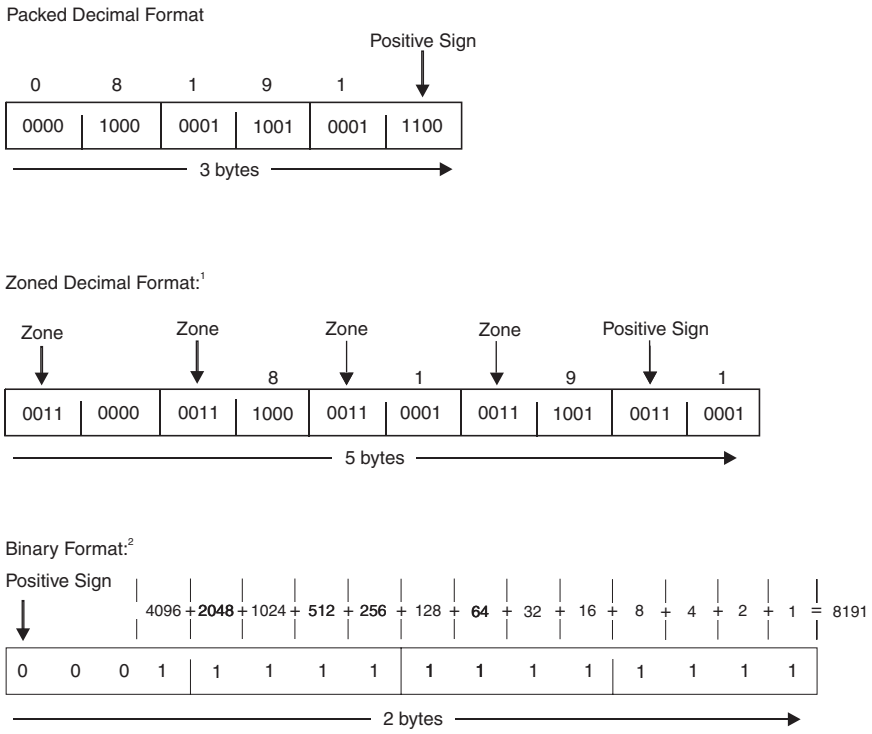


Figure 37. Defining Binary Fields

[1]If 8191 is read into storage as a zoned-decimal field, it occupies 4 bytes. If it is converted to packed-decimal format, it occupies 3 bytes. When it is converted back to zoned-decimal format, it occupies 5 bytes.

[2]To obtain the numeric value of a positive binary number add the values of the bits that are on (1), do not include the sign bit. To obtain the numeric value of a negative binary number, add the values of the bits that are off (0) plus one (the sign bit is not included).

Float Format

The float format consists of two parts:
- the mantissa
- the exponent

The value of a floating-point field is the result of multiplying the mantissa by 10 raised to the power of the exponent. For example, if 1.2345 is the mantissa and 5 is the exponent then the value of the floating-point field is:

```
1.2345 * (10 ** 5) = 123450
```

You define a floating-point field by specifying F in the data type entry of the appropriate specification.

The decimal positions must be left blank. However, floating-point fields are considered to have decimal positions. As a result, float variables may not be used in any place where a numeric value without decimal places is required, such as an array index, do loop index, and so on.

The default initialization and CLEAR value for a floating point field is 0E0.

The length of a floating point field is defined in terms of the number of bytes. It must be specified as either 4 or 8 bytes. The range of values allowed for a positive floating-point field are:

Field length	Minimum Allowed Value	Maximum Allowed Value
4 bytes	1.175 494 4 E-38	3.402 823 5 E+38
8 bytes	2.225 073 858 507 201 E-308	1.797 693 134 862 315 E+308

Note: Negative values have the same range, but with a negative sign.

Since float variables are intended to represent ″scientific″ values, a numeric value stored in a float variable may not represent the exact same value as it would in a packed variable. Float should not be used when you need to represent numbers exactly to a specific number of decimal places, such as monetary amounts.

External Display Representation of a Floating-Point Field

See “Specifying an External Format for a Numeric Field” on page 104 for a general description of external display representation.

The external display representation of float values applies for the following:
- Output of float data with Data-Format entry blank.
- Input of float data with Data-Format entry blank.
- External format of compile-time and prerun-time arrays and tables (when keyword EXTFMT is omitted).
- Display and input of float values using operation code DSPLY.
- Result of built-in function %EDITFLT.

Output: When outputting float values, the external representation uses a format similar to float literals, except that:
- Values are always written with the character **E** and the signs for both mantissa and exponent.
- Values are either 14 or 23 characters long (for **4F** and **8F** respectively).
- Values are normalized. That is, the decimal point immediately follows the most significant digit.
- The decimal separator character is either period or comma depending on the parameter for Control-Specification keyword DECEDIT.

Here are some examples of how float values are presented:

```
+1.2345678E-23
-8.2745739E+03
-5.722748027467392E-123
+1,2857638E+14            if DECEDIT(',') is specified
```

Input: When inputting float values, the value is specified just like a float literal. The value does not have to be normalized or adjusted in the field. When float values are defined as array/table initialization data, they are specified in fields either 14 or 23 characters long (for **4F** and **8F** respectively).

Note the following about float fields:
- Alignment of float fields may be desired to improve the performance of accessing float subfields. You can use the ALIGN keyword to align float subfields defined on a definition specification. 4-byte float subfields are aligned on a 4-byte boundary and 8-byte float subfields are aligned along a 8-byte boundary. For more information on aligning float subfields, see “ALIGN” on page 265.

- Length adjustment is not allowed when the LIKE keyword is used to define a field like a float field.

Integer Format

The integer format is similar to the binary format with two exceptions:
- The integer format allows the full range of binary values
- The number of decimal positions for an integer field is always zero.

You define an integer field by specifying I in the Data-Type entry of the appropriate specification. You can also define an integer field using the LIKE keyword on a definition specification where the parameter is an integer field.

The length of an integer field is defined in terms of number of digits; it can be 3, 5, 10, or 20 digits long. A 3-digit field takes up 1 byte of storage; a 5-digit field takes up 2 bytes of storage; a 10-digit field takes up 4 bytes; a 20-digit field takes up 8 bytes. The range of values allowed for an integer field depends on its length.

Field length
Range of Allowed Values

3-digit integer
-128 to 127

5-digit integer
-32768 to 32767

10-digit integer
-2147483648 to 2147483647

20-digit integer
-9223372036854775808 to 9223372036854775807

Note the following about integer fields:
- Alignment of integer fields may be desired to improve the performance of accessing integer subfields. You can use the ALIGN keyword to align integer subfields defined on a definition specification.

  2-byte integer subfields are aligned on a 2-byte boundary and 4-byte integer subfields are aligned along a 4-byte boundary; 8-byte integer subfields are aligned along an 8-byte boundary. For more information on aligning integer subfields, see "ALIGN" on page 265.
- If the LIKE keyword is used to define a field like an integer field, the Length entry may contain a length adjustment in terms of number of digits. The adjustment value must be such that the resulting number of digits for the field is 3, 5, 10, or 20.

Packed-Decimal Format

Packed-decimal format means that each byte of storage (except for the low-order byte) can contain two decimal numbers. The low-order byte contains one digit in the leftmost portion and the sign (positive or negative) in the rightmost portion. All packed-decimal numbers use the preferred signs: hexadecimal C for positive numbers and hexadecimal D for negative numbers. In addition, the following signs are supported: hexadecimal A, E, F for positive numbers and hexadecimal B for negative numbers. The packed-decimal format looks like this:

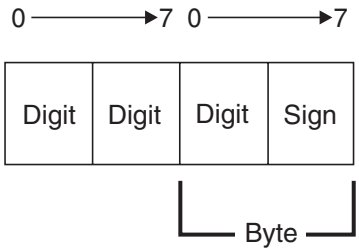


Figure 37 on page 124 shows what the decimal number 8191 looks like in packed-decimal format.

For a program-described file:

- Specify P in position 36 of the input specifications for packed-decimal input

  Specify P in position 52 of the output specifications for packed-decimal output. This position must be blank if editing is specified.

  Specify P in position 40 of the definition specifications for packed-decimal arrays and tables. The external format for compile-time arrays and tables cannot be packed-decimal format.

For an externally described file, the data format is specified in the data description specifications.

Determining the Digit Length of a Packed-Decimal Field

Use the following formula to find the length in digits of a packed-decimal field:

```
Number of digits = 2n - 1,
...where n = number of packed input record positions used.
```

This formula gives you the maximum number of digits you can represent in packed-decimal format; the upper limit is 31.

Packed fields can be up to 16 bytes long. Table 17 shows the packed equivalents for zoned-decimal fields up to 31 digits long:

Table 17. Packed Equivalents for Zoned-Decimal Fields up to 31 Digits Long

Zoned-Decimal Length in Digits	Number of Bytes Used in Packed-Decimal Field
1	1
2, 3	2
4, 5	3
. . . . . .	. . .
28, 29	15
30	16
31	16

For example, an input field read in packed-decimal format has a length of five bytes (as specified on the input or data description specifications). The number of digits in this field equals 2(5) – 1 or 9. Therefore, when the field is used in the calculation specifications, the result field must be nine positions long. The PACKEVEN keyword on the definition specification can be used to indicate which

of the two possible sizes you want when you specify a packed subfield using from and to positions rather than number of digits.

Unsigned Format

The unsigned integer format is like the integer format except that the range of values does not include negative numbers. You should use the unsigned format only when non-negative integer data is expected.

You define an unsigned field by specifying U in the Data-Type entry of the appropriate specification. You can also define an unsigned field using the LIKE keyword on the definition specification where the parameter is an unsigned field.

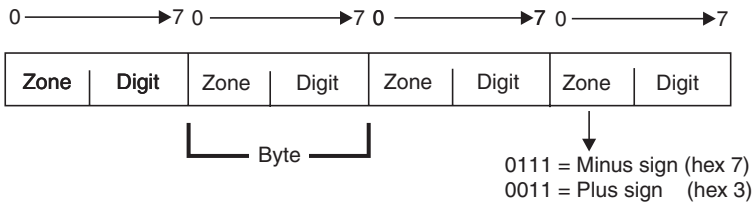
The length of an unsigned field is defined in terms of number of digits; it can be 3, 5, 10, or 20 digits long. A 3-digit field takes up 1 byte of storage; a 5-digit field takes up 2 bytes of storage; a 10-digit field takes up 4 bytes;a 20-digit field takes up 8 bytes. The range of values allowed for an unsigned field depends on its length.

Field length	Range of Allowed Values
3-digit unsigned	0 to 255
5-digit unsigned	0 to 65535
10-digit unsigned	0 to 4294967295
20-digit unsigned	0 to 18446744073709551615

For other considerations regarding the use of unsigned fields, including information on alignment, see “Integer Format” on page 126.

Zoned-Decimal Format

Zoned-decimal format means that each byte of storage can contain one digit or one character. In the zoned-decimal format, each byte of storage is divided into two portions: a 4-bit zone portion and a 4-bit digit portion. The zoned-decimal format looks like this:



The zone portion of the right-most byte indicates the sign (positive or negative) of the decimal number. All zoned-decimal numbers use the preferred signs: hexadecimal 3 for positive numbers and hexadecimal 7 for negative numbers. In addition, the following signs are supported: hexadecimal 0, 1, 2, 8, 9, A, B for positive numbers and hexadecimal 4, 5, 6, C, D, E, F for negative numbers. In zoned-decimal format, each digit in a decimal number includes a zone portion; however, only the right-most zone portion serves as the sign. Figure 37 on page 124 shows what the number 8191 looks like in zoned-decimal format.

You must consider the change in field length when coding the end position in positions 40 through 43 of the output specifications and the field is to be output in packed format. To find the length of the field after it has been packed, use the following formula:

$$\text{Field length} = \frac{n}{2} + 1$$

. . . where n = number of digits in the zoned decimal field.

(Any remainder from the division is ignored.)

For a program-described file, zoned-decimal format is specified by a blank in position 36 of the input specifications, in position 52 of the output specifications, or in position 40 of the definition specifications. For an externally described file, the data format is specified in position 35 of the data description specifications.

You can specify an alternative sign format for zoned-decimal format. In the alternative sign format, the numeric field is immediately preceded or followed by a + or − sign. A plus sign is a hexadecimal 2B, and a minus sign is a hexadecimal 2D.

When an alternative sign format is specified, the field length (specified on the input specification) must include an additional position for the sign. For example, if a field is 5 digits long and the alternative sign format is specified, a field length of 6 positions must be specified.

Considerations for Using Numeric Formats

Keep in mind the following when defining numeric fields:

- When coding the end position in positions 47 through 51 of the output specifications, be sure to use the external format when calculating the number of bytes to be occupied by the output field. For example, a packed field with 5 digits is stored in 3 bytes, but when output in zoned format, it requires 5 bytes. When output in integer format, it only requires 2 bytes.
- If you move a character field to a zoned numeric, the sign of the character field is fixed to zoned positive or zoned negative. The zoned portion of the other bytes will be forced to '3'. However, if the digit portion of one of the bytes in the character field does not contain a valid digit a decimal data error will occur.
- When numeric fields are written out with no editing, the sign is not printed as a separate character; the last digit of the number will include the sign. This can produce surprising results; for example, when -625 is written out, the zoned decimal value is XX'363275' which appears as 62u.
- The default is to perform 4-byte arithmetic. The compiler only performs 8-byte arithmetic if at least one operand is an 8-byte integer. An overflow runtime error can occur for those arithmetic operations where two 4-byte integers produce an 8-byte result. To avoid this problem, make sure one operand is 8 bytes.

Guidelines for Choosing the Numeric Format for a Field

You should specify the integer or unsigned format for fields when:

- Performance of arithmetic is important

  With certain arithmetic operations, it may be important that the value used be an integer. Some examples where performance may be improved include array index computations and arguments for the built-in function %SUBST.
- The default is to perform 4-byte arithmetic. The compiler only performs 8-byte arithmetic if at least one operand is an 8-byte integer. From a performance perspective, 8-byte arithmetic is expensive and should be avoided.
- Interacting with routines written in other languages that support an integer data type, such as ILE C.
- Using fields in file feedback areas that are defined as integer and that may contain values above 9999 or 999999999.

Packed, zoned, and binary formats should be specified for fields when:

- Using values that have implied decimal positions, such currency values
- Manipulating values having more than 19 digits
- Ensuring a specific number of digits for a field is important

Float format should be specified for fields when:

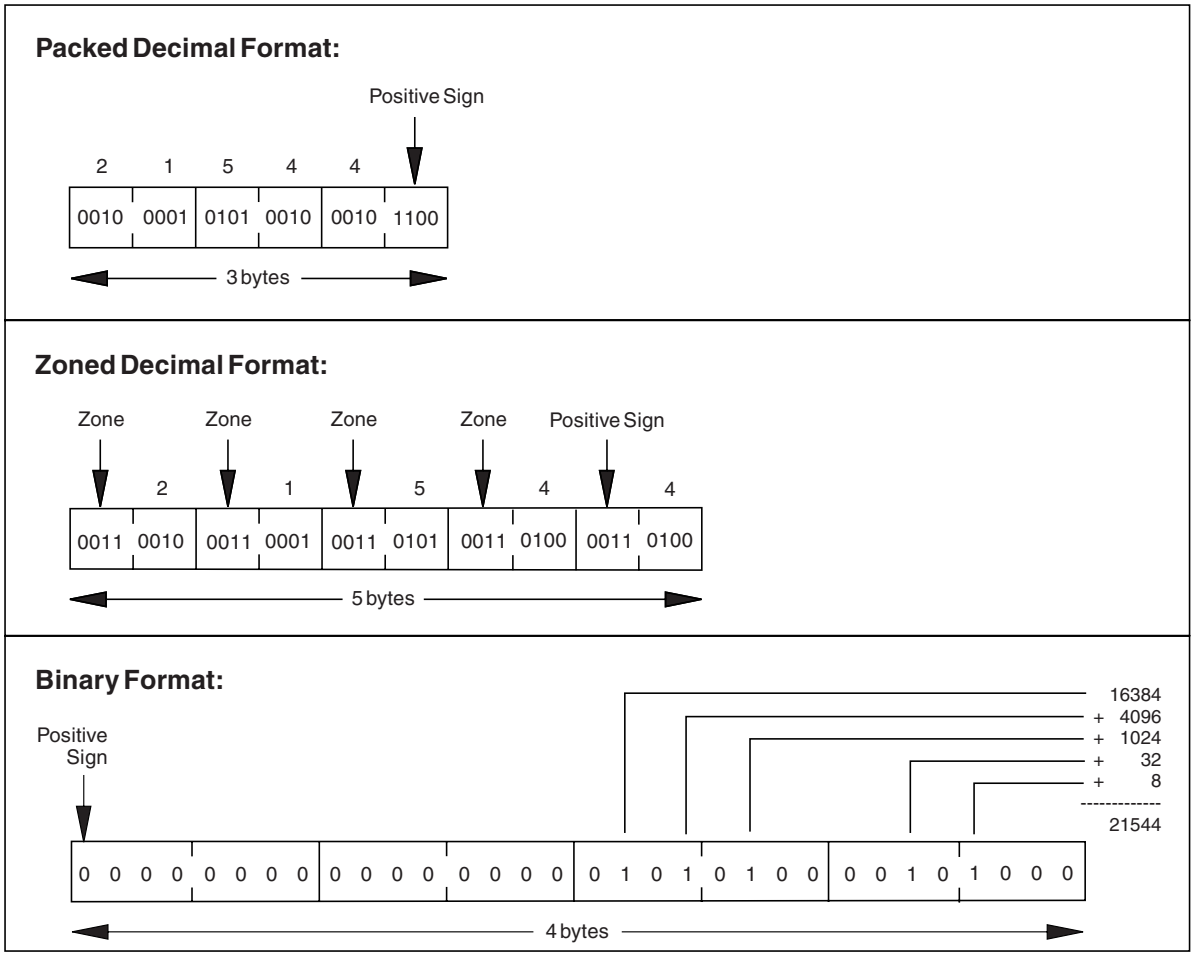
- The same variable is needed to hold very small and/or very large values that cannot be represented in packed or zoned values.

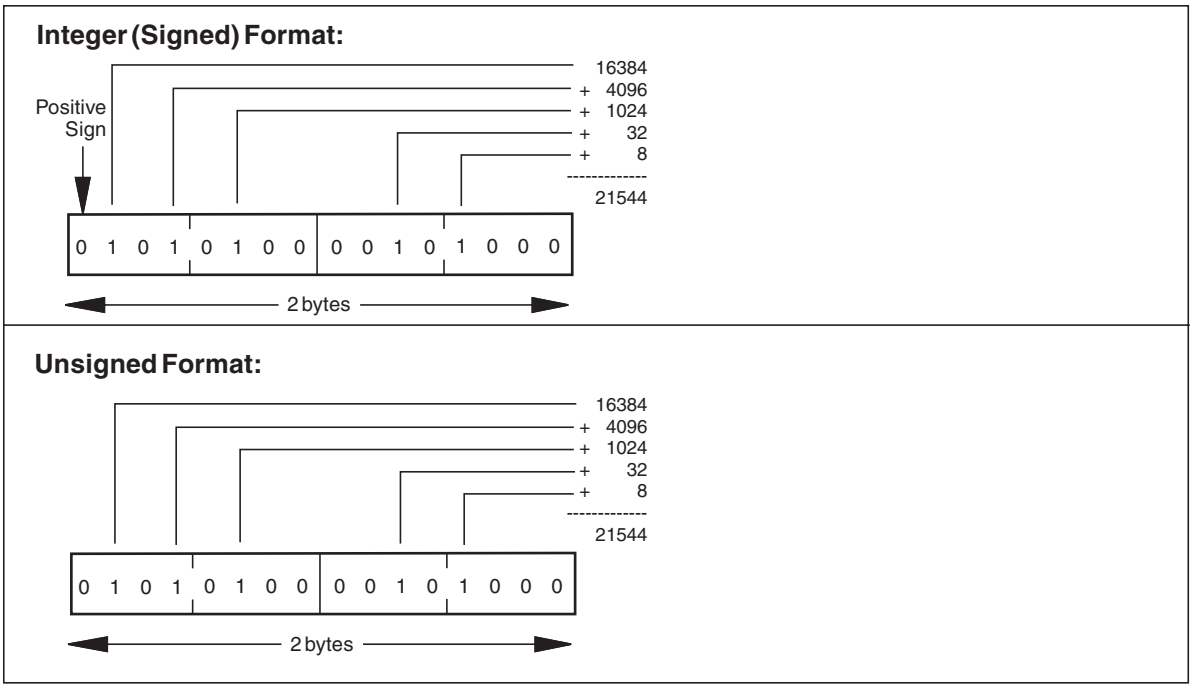
However, float format should *not* be used when more than 16 digits of precision are needed.

Note: Overflow is more likely to occur with arithmetic operations performed using the integer or unsigned format, especially when integer arithmetic occurs in free-form expressions. This is because the intermediate results are kept in integer or unsigned format rather than a temporary decimal field of sufficient size.

Representation of Numeric Formats

The following figure shows what the decimal number 21544 looks like in various formats.





Note the following about the representations in the figure.

- To obtain the numeric value of a positive binary or integer number, unsigned number, add the values of the bits that are on (1), but do not include the sign bit (if present). For an unsigned number, add the values of the bits that are on, including the leftmost bit.
- The value 21544 cannot be represented in a 2-byte binary field even though it only uses bits in the low-order two bytes. A 2-byte binary field can only hold up to 4 digits, and 21544 has 5 digits.

Figure 38 shows the number -21544 in integer format.

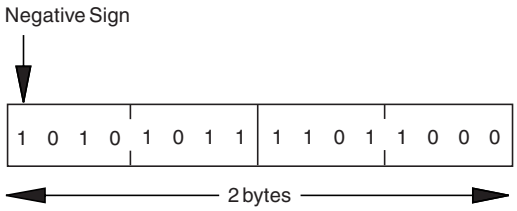


Figure 38. Integer Representation of the Number -21544

Note: The workstation architecture stores binary, integer, and unsigned formats in program memory in a byte-reversed order. This storage mechanism will affect the value of any character subfields used to overlay subfields for these formats.

Object Data Type

The object data type allows you to define a Java object. You specify the object data type as follows:

```
 * Variable MyString is a Java String object.
D MyString        S               O   CLASS(*JAVA
D                                         :'java.lang.String')
```

or as follows:

```
D bdcreate        PR              O   EXTPROC(*JAVA
D                                           :'java.math.BigDecimal'
D                                           :*CONSTRUCTOR)
```

In position 40, you specify data type O. In the keyword section, you specify the CLASS keyword to indicate the class of the object. Specify *JAVA for the environment, and the class name.

If the object is the return type of a Java constructor, the class of the returned object is the same as the class of the method so you do not specify the CLASS keyword. Instead, you specify the EXTPROC keyword with environment *JAVA, the class name, and procedure name *CONSTRUCTOR.

An object cannot be based. It also cannot be a subfield of a data structure.

If an object is an array or table, it must be loaded at runtime. Pre-run and compile-time arrays and tables of type Object are not allowed.

Every object is initialized to *NULL, which means that the object is not associated with an instance of its class.

To change the contents of an object, you must use method calls. You cannot directly access the storage used by the object.

Classes are resolved at runtime. The compiler does not check that a class exists or that it is compatible with other objects.

Where You Can Specify an Object Field

You can use an object field in the following situations:

Free-Form Evaluation

You can use the EVAL operation to assign one Object item (field or prototyped procedure) to a field of type Object.

Free-Form Comparison

You can compare one object to another object. You can specify any comparison, but only the following comparisons are meaningful:

- Equality or inequality with another object. Two objects are equal only if they represent exactly the same object. Two different objects with the same value are not equal.

  If you want to test for equality of the value of two objects, use the Java 'equals' method as follows:

```
D objectEquals    PR              N   EXTPROC(*JAVA
D                                         : 'java.lang.Object'
D                                         : 'equals')
C                   IF        objectEquals (obj1 : obj2)
C                    ...
C                   ENDIF
```

- Equality or inequality with *NULL. An object is equal to *NULL if it is not associated with a particular instance of its class.

Free-Form Call Parameter

You can code an object as a parameter in a call operation if the parameter in the prototype is an object.

Notes:

1. Objects are not valid as input or output fields.
2. Assignment validity is not checked. For example, RPG would allow you to assign an object of class Number to an object variable defined with class String. If this was not correct, a Java error would occur when you tried to use the String variable.

```
D Obj             S               O   CLASS(*JAVA
D                                         :'java.lang.Object')
D Str             S               O   CLASS(*JAVA
D                                         :'java.lang.String')
D Num             S               O   CLASS(*JAVA
D                                         :'java.math.BigDecimal')

 * Since all Java classes are subclasses of class 'java.lang.Object',
 * any object can be assigned to a variable of this class.
 * The following two assignments are valid.
C                   EVAL      Obj = Str
C                   EVAL      Obj = Num
 * However, it would probably not be valid to assign Str to Num.
```

Figure 39. Object Data Type Example

Procedure Pointer Data Type

Procedure pointers are used to point to procedures or functions. A procedure pointer points to an entry point that is bound into the program. Procedure pointers are defined on the definition specification.

The length of the procedure pointer field must be 4 bytes long and must be aligned on a 4 byte boundary. This requirement for boundary alignment can cause a pointer subfield of a data structure not to follow the preceding field directly, and

can cause multiple occurrence data structures to have non-contiguous occurrences. The default initialization value for procedure pointers is *NULL.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D*
D* Define a basing pointer field and initialize to the address of the
D* data structure My_Struct.
D*
D  My_struct      DS
D   My_array                    10    DIM(50)
D
D Ptr1            S              4*   INZ(%ADDR(My_Struct))
D*
D* Or equivalently, defaults to length 4 if length not defined
D*
D Ptr1            S               *   INZ(%ADDR(My_Struct))
D*
D* Define a procedure pointer field and initialize to NULL
D*
D Ptr1            S              4*   PROCPTR INZ(*NULL)
D*
D* Define a procedure pointer field and initialize to the address
D* of the procedure My_Proc.
D*
D Ptr1            S              4*   PROCPTR INZ(%PADDR(My_Proc))
D*
D* Define pointers in a multiple occurrence data structure and map out
D* the storage.
D*
DDataS            DS                  OCCURS(2)
D ptr1                            *
D ptr2                            *
D Switch                         1A
D*
D* Storage map would be:
D*
D*               DataS
```

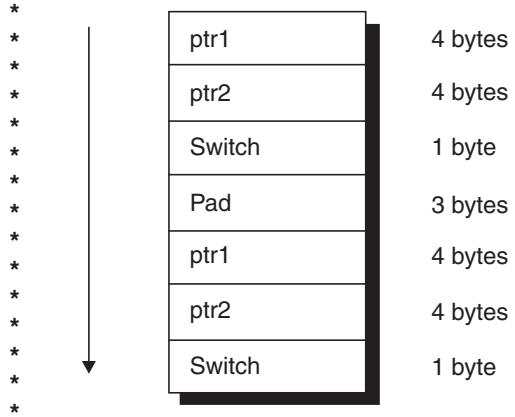


Figure 40. Defining Pointers

Time Data

Time fields have a predetermined size and format. They can be defined on the definition specification. Leading and trailing zeros are required for all time data.

Time constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Times are used for I/O operations where input fields, output fields or key fields are converted (if required) to the necessary format for the operation.

The default internal format for time variables is *ISO. This default internal format can be overridden globally by the control specification keyword TIMFMT and individually by the definition specification keyword TIMFMT.

The hierarchy used when determining the internal time format and separator for a time field is:
1. From the TIMFMT keyword specified on the definition specification
2. From the TIMFMT keyword specified on the control specification
3. *ISO

For examples on how to code time fields, see the examples in:
- "Date Operations" on page 359
- "Moving Date-Time Data" on page 370
- "ADDDUR (Add Duration)" on page 502
- "MOVE (Move)" on page 604
- "SUBDUR (Subtract Duration)" on page 693
- "TEST (Test Date/Time/Timestamp)" on page 700

The following table lists the formats for time data:

Table 18. Time Formats and Separators for Time data type

Format Name	Description	Format with Default Separator)	Valid Sepa-rators	Length	Example
*HMS	Hours:Minutes:Seconds	hh:mm:ss	: . , &	8	14:00:00
*ISO	International Standards Organization	hh.mm.ss	.	8	14.00.00
*USA	IBM USA Standard. AM and PM can be any mix of upper and lower case.	hh:mm AM or hh:mm PM	:	8	02:00 PM
*EUR	IBM European Standard	hh.mm.ss	.	8	14.00.00
*JIS	Japanese Industrial Standard Christian Era	hh:mm:ss	:	8	14:00:00

The following table lists the *LOVAL, *HIVAL, and default values for all the date formats:

Table 19. Time Values

Format name	Description	*LOVAL	*HIVAL	Default Value
*HMS	Hours:Minutes:Seconds	00:00:00	24:00:00	00:00:00
*ISO	International Standards Organization	00.00.00	24.00.00	00.00.00
*USA	IBM USA Standard. AM and PM can be any mix of upper and lower case.	00:00 AM	12:00 AM	00:00 AM
*EUR	IBM European Standard	00.00.00	24.00.00	00.00.00
*JIS	Japanese Industrial Standard Christian Era	00:00:00	24:00:00	00:00:00

Separators

When coding a time format on a MOVE, MOVEL or TEST operation, separators are optional for character fields. To indicate that there are no separators, specify the format followed by a zero. For more information on how to code time formats without separators see "MOVE (Move)" on page 604.

Timestamp Data

Timestamp fields have a predetermined size and format. They can be defined on the definition specification. Timestamp data must be in the format

```
yyyy-mm-dd-hh.mm.ss.mmmmmm (length 26).
```

Microseconds (.mmmmmm) are optional for timestamp literals and if not provided will be padded on the right with zeroes. Leading zeros are required for all timestamp data.

The default initialization value for a timestamp is midnight of January 1, 0001 (0001-01-01-00.00.00.000000). The *HIVAL value for a timestamp is 9999-12-31-24.00.00.000000. Similarly, the *LOVAL value for timestamp is 0001-01-01-00.00.00.00000.

Separators

When coding the timestamp format on a MOVE, MOVEL or TEST operation, separators are optional for character fields. To indicate that there are no separators, specify *ISO0. For an example of how *ISO is used without separators see "TEST (Test Date/Time/Timestamp)" on page 700.

Database Null Value Support

In a VisualAge RPG program, you can select one of three different ways of handling null-capable fields from an externally-described database file. This depends on how you specify the *Allow null values* option or ALWNULL control specification keyword:

1. **User control**, ALWNULL(*USRCTL) - read, write, update, and delete records with null values and retrieve and position-to records with null keys.
2. **Input only**, ALWNULL(*INPUTONLY) - read records with null values to access the data in the null fields
3. **No**, ALWNULL(*NO)- do not process records with null values

Note: For a program-described file, a null value in the record always causes a data mapping error, regardless of the value specified on the *Allow null values* option or ALWNULL keyword

For more information on specifying compiler options, *Getting Started with WebSphere Development Studio Client for iSeries*, SC09-2625-06.

User Controlled Support for Null-Capable Fields and Key Fields

When an externally-described file contains null-capable fields and the **User control** or ALWNULL(*USRCTL) option is specified, you can do the following:

- Read, write, update, and delete records with null values from externally-described database files.
- Retrieve and position-to records with null keys using keyed operations, by specifying an indicator in factor 2 of the KFLD associated with the field.
- Determine whether a null-capable field is actually null using the %NULLIND built-in function on the right-hand-side of an expression.
- Set a null-capable field to be null for output or update using the %NULLIND built-in function on the left-hand-side of an expression.

You are responsible for ensuring that fields containing null values are used correctly within the program. For example, if you use a null-capable field as factor 2 of a MOVE operation, you should first check if it is null before you do the MOVE, otherwise you may corrupt your result field value. You should also be careful when outputting a null-capable field to a file that does not have the field defined as null-capable, for example a PRINTER or a program-described file.

Note: The value of the null indicator for a null-capable field is only considered for these operations: input, output and file-positioning. Here are some examples of operations where the null indicator is not taken into consideration:
- DSPLY of a null-capable field shows the contents of the field even if the null indicator is on.
- If you move a null-capable field to another null-capable field, and the factor 2 field has the null indicator on, the result field will get the data from the factor 2 field. The corresponding null indicator for the result field will not be set on.
- Comparison operations, including SORTA and LOOKUP, with null capable fields do not consider the null indicators.

A field is considered null-capable if it is null-capable in any externally-described database record and is not defined as a constant in the program.

When a field is considered null-capable in a VARPG program, a null indicator is associated with the field. Note the following:

- If the field is a multiple-occurrence data structure or a table, an array of null indicators will be associated with the field. Each null indicator corresponds to an occurrence of the data structure or element of the table.
- If the field is an array element, the entire array will be considered null-capable. An array of null indicators will be associated with the array, each null indicator corresponds to an array element.
- If the field is an element of an array subfield of a multiple-occurrence data structure, an array of null indicators will be associated with the array for each occurrence of the data structure.

Null indicators are initialized to zeros during program initialization and thus null-capable fields do not contain null values when the program starts execution.

Null-capable fields in externally-described data structures

If the file used for an externally described data structure has null-capable fields defined, the matching RPG subfields are defined to be null-capable. Similarly, if a record format has null-capable fields, a data structure defined with LIKEREC will have null-capable subfields. When a data structure has null-capable subfields, another data structure defined like that data structure using LIKEDS will also have null-capable subfields. However, using the LIKE keyword to define one field like another null-capable field does not cause the new field to be null-capable.

Input of Null-Capable Fields

For a field that is null-capable in the RPG program, the following will apply on input, for DISK and SPECIAL files:

- When a null-capable field is read from an externally-described file, the null indicator for the field is set on if the field is null in the record. Otherwise, the null indicator is set off.
- If field indicators are specified and the null-capable field is null, all the field indicators will be set off.
- If a field is defined as null-capable in one file, and not null-capable in another, then the field will be considered null-capable in the RPG program. However, when you read the second file, the null indicator associated with the field will always be set off.
- An input operation from a program-described file using a data structure in the result field does not affect the null indicator associated with the data structure or any of its subfields.
- Reading null-capable fields using input specifications for program-described files always sets off the associated null indicators.
- If null-capable fields are not selected to be read due to a field-record-relation indicator, the associated null indicator will not be changed.
- When a record format or file with null-capable fields is used on an input operation (READ, READP, READE, READPE, CHAIN) and a data structure is coded in the result field, the values of %NULLIND for null-capable data structure subfields will be changed by the operation. The values of %NULLIND will not be set for the input fields for the file, unless the input fields happen to be the subfields used in the input operation.

Output of Null-Capable Fields

When a null-capable field is written (output or update) to an externally-described file, a null value is written out if the null indicator for the field is on at the time of the operation.

When a null-capable field is output to or updated in an externally-described database file, then if the field is null, the value placed in the buffer will be ignored by data management.

Note: Fields that have the null indicator on at the time of output have the data moved to the buffer. This means that errors such as decimal-data error, or basing pointer not set, will occur even if the null indicator for the field is on.

During an output operation to an externally-described database file, if the file contains fields that are considered null-capable in the program but not null-capable in the file, the null indicators associated with those null-capable fields will not be used.

When a record format with null-capable fields is used on a WRITE or UPDATE operation, and a data structure is coded in the result field, the null attributes of the data structure subfields will be used to set the null-byte-map for the output or update record.

When a record format with null-capable fields is used on an UPDATE operation with %FIELDS, then the null-byte-map information will be taken from the null attributes of the specified fields.

Figure 41 shows how to read, write and update records with null values when the **User control** option or ALWNULL(*USRCTL) keyword is selected.

```
H*
H* Specify the ALWNULL(*USRCTL) keyword on a control
H* specification or compile the VARPG program with the
H* User control option.
H*
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H*      H ALWNULL(*USRCTL)
F*
F* DISKFILE contains a record REC which has 2 fields: FLD1 and FLD2
F* Both FLD1 and FLD2 are null-capable.
F*
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
F*
FDISKFILE  UF A E             DISK
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
C*
C* Read the first record.
C* Update the record with new values for any fields which are not
C* null.
C                   READ      REC                                    10
C                   IF        NOT %NULLIND(Fld1)
C                   MOVE      'FLD1'        Fld1
C                   ENDIF
C                   IF        NOT %NULLIND(Fld2)
C                   MOVE      'FLD2'        Fld2
C                   ENDIF
C                   UPDATE    REC
C*
C* Read another record.
C* Update the record so that all fields are null.
C* There is no need to set the values of the fields because they
C* would be ignored.
C                   READ      REC                                    10
C                   EVAL      %NULLIND(Fld1) = *ON
C                   EVAL      %NULLIND(Fld2) = *ON
C                   UPDATE    REC
C*
C* Write a new record where Fld 1 is null and Fld 2 is not null.
C*
C                   EVAL      %NULLIND(Fld1) = *ON
C                   EVAL      %NULLIND(Fld2) = *OFF
C                   EVAL      Fld2 = 'New value'
C                   WRITE     REC
```

Figure 41. Input and Output of Null-Capable Fields

Keyed Operations

If you have a null-capable key field, you can search for records containing null values by specifying an indicator in factor 2 of the KFLD operation and setting that indicator on before the keyed input operation. If you do not want a null key to be selected, you set the indicator off.

When a record format with null-capable key fields is used on a CHAIN, SETLL, READE, or READPE operation, and a %KDS data structure is used to specify the keys, then the null-key-byte-map information will be taken from the null attributes of the subfields in the data structure specified as the argument of %KDS.

When a record format with null-capable key fields is used on a CHAIN, SETLL, READE, or READPE operation, and a list of keyfields is used, then the null-key-byte-map information will be taken from the null attributes of the specified keys.

Figure 42 on page 142 and Figure 43 on page 143 illustrate how keyed operations are used to position and retrieve records with null keys.

```
|    // Assume File1 below contains a record Rec1 with a composite key
|    // made up of three key fields: Key1, Key2, and Key3.  Key2 and Key3
|    // are null-capable.  Key1 is not null-capable.
|    // Each key field is two characters long.
|  *..1....+....2....+....3....+....4....+....5....+....6....+....7....+..
|  FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
|  FFile1     IF   E             DISK      REMOTE
|    // Define two data structures with the keys for the file
|    // Subfields Key2 and Key3 of both data structures will be
|    // null-capable.
|  DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
|  D Keys           DS                  LIKEREC(Rec1 : *KEY)
|  D OtherKeys      DS                  LIKEDS(keys)
|    // Define a data structure with the input fields of the file
|    // Subfields Key2 and Key3 of the data structures will be
|    // null-capable.
|  D File1Flds      DS                  LIKEREC(Rec1 : *INPUT)
|   /free
|       // The null indicator for Keys.Key2 is ON and the
|       // null indicator for Keys.Key3 is OFF, for the
|       // SETLL operation below.  File1 will be positioned
|       // at the next record that has a key that is equal
|       // to or greater than 'AA??CC' (where ?? is used
|       // in this example to indicate NULL)
|
|       // Because %NULLIND(Keys.Key2) is ON, the actual content
|       // in the search argument Keys.Key2 will be ignored.
|
|       // If a record exists in File1 with 'AA' in Key1, a null
|       // Key2, and 'CC' in Key3, %EQUAL(File1) will be true.
|
|       Keys.Key1 = 'AA';
|       Keys.Key3 = 'CC';
|       %NULLIND(Keys.Key2) = *ON;
|       %NULLIND(Keys.Key3) = *OFF;
|       SETLL %KDS(Keys) Rec1;
|       // The CHAIN operation below will retrieve a record
|       // with 'JJ' in Key1, 'KK' in Key2, and a null Key3.
|       // Since %NULLIND(OtherKeys.Key3) is ON, the value of
|       // 'XX' in OtherKeys.Key3 will not be used.  This means
|       // that if File1 actually has a record with a key
|       // 'JJKKXX', that record will not be retrieved.
|
|       OtherKeys.Key3 = 'XX';
|       %NULLIND(Keys.Key3) = *ON;
|       CHAIN ('JJ' : 'KK' : OtherKeys.Key3) Rec1;
|       // The CHAIN operation below uses a partial key as the
|       // search argument.  It will retrieve a record with 'NN'
|       // in Key1, a null key2, and any value including a null
|       // value in Key3.  The record is retrieved into the
|       // File1Flds data structure, which will cause the
|       // null flags for File1Flds.Key2 and File1Flds.Key3
|       // to be changed by the operation (if the CHAIN
|       // finds a record).
|
|       Keys.Key1 = 'NN';
|       %NULLIND(Keys.Key2) = *ON;
|       CHAIN %KDS(Keys : 2) Rec1 File1Flds;
|
```

Figure 42. Example of handling null-capable key fields

```
 * Using the same file as the previous example, define two
 * key lists, one containing three keys and one containing
 * two keys.
 *
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
C*
C     Full_Kl       KLIST
C                   KFLD                    Key1
C                   KFLD      *IN02         Key2
C                   KFLD      *IN03         Key3
C*
C     Partial_Kl    KLIST
C                   KFLD                    Key1
C                   KFLD      *IN05         Key2
C*
C*
C* *IN02 is ON and *IN03 is OFF for the SETLL operation below.
C* File1 will be positioned at the next record that has a key
C* that is equal to or greater than 'AA??CC' (where ?? is used
C* in this example to indicate NULL)
C*
C* Because *IN02 is ON, the actual content in the search argument
C* for Key2 will be ignored.
C*
C* If a record exists in File1 with 'AA' in Key1, a null Key2, and
C* 'CC' in Key3, indicator 90 (the Eq indicator) will be set ON.
C*
C                   MOVE      'AA'          Key1
C                   MOVE      'CC'          Key3
C                   EVAL      *IN02 = '1'
C                   EVAL      *IN03 = '0'
C     Full_Kl       SETLL     Rec1                                  90
C*
```

Figure 43. Example of Keyed Operations Using Null-Capable Key Fields (Part 1 of 2)

```
C*
C* The CHAIN operation below will retrieve a record with 'JJ' in Key1,
C* 'KK' in Key2, and a null Key3.  Again, because *IN03 is ON, even
C* if the programmer had moved some value (say 'XX') into the search
C* argument for Key3, 'XX' will not be used.  This means if File1
C* actually has a record with a key 'JJKKXX', that record will not
C* be retrieved.
C*
C                   MOVE      'JJ'          Key1
C                   MOVE      'KK'          Key2
C                   EVAL      *IN02 = '0'
C                   EVAL      *IN03 = '1'
C     Full_Kl       CHAIN     Rec1                              80
C*
C*
C* The CHAIN operation below uses a partial key as the search argument.
C* It will retrieve a record with 'NN' in Key1, a null key2, and any
C* value including a null value in Key3.
C*
C* In the database, the NULL value occupies the highest position in
C* the collating sequence.  Assume the keys in File1 are in ascending
C* sequence.  If File1 has a record with 'NN??xx' as key (where ??
C* means NULL and xx means any value other than NULL), that record
C* will be retrieved.  If such a record does not exist in File1, but
C* File1 has a record with 'NN????' as key, the 'NN????' record will
C* be retrieved.  The null flags for Key2 and Key3 will be set ON
C* as a result.
C*
C                   MOVE      'NN'          Key1
C                   SETON                                       05
C     Partial_Kl    CHAIN     Rec1                              70
```

Figure 43. Example of Keyed Operations Using Null-Capable Key Fields (Part 2 of 2)

Input-Only Support for Null-Capable Fields

When an externally-described input-only file contains null-capable fields and the **Input only** option or ALWNULL(*INPUTONLY) keyword is specified, the following conditions apply:

- When a record is retrieved from a database file and there are some fields containing null values in the record, database default values for the null-capable fields will be placed into those fields containing null values. The default value will be the user defined DDS defaults or system defaults.
- You will not be able to determine whether any given field in the record has a null value.
- Field indicators are not allowed on an input specification if the input field is a null-capable field from an externally-described input-only file.
- Keyed operations are not allowed when factor 1 of a keyed input calculation operation corresponds to a null-capable key field in an externally-described input-only file.

No Null Fields Option

When an externally-described file contains null-capable fields and the **No** option or ALWNULL(*NO) keyword is specified, the following conditions apply:

- A record containing null values retrieved from a file will cause a data mapping error and an error message will be issued.
- Data in the record is not accessible and none of the fields in the record can be updated with the values from the input record containing null values.

- With this option, you cannot place null values in null-capable fields for updating or adding a record. If you want to place null values in null-capable fields, use the **User control** option.

Converting Database Variable-Length Fields

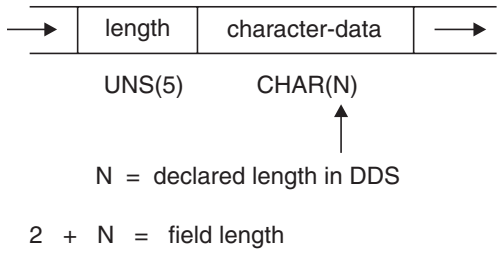
The VisualAge RPG compiler can internally define variable-length character or graphic fields from an externally described file or data structure as fixed-length character fields. Although converting variable-length character and graphic fields to fixed-length format is not necessary, the CVTOPT compiler option remains in the language to support programs written before variable-length fields were supported.

You can convert variable-length fields by specifying *VARCHAR (for variable-length character fields) or *VARGRAPHIC (for variable-length graphic fields) on the CVTOPT control specification keyword. When *VARCHAR or *VARGRAPHIC is not specified, or *NOVARCHAR or *NOVARGRAPHIC is specified, variable-length fields are not converted to fixed-length character and can be used in your VisualAge RPG program as variable-length.

The following conditions apply when *VARCHAR or *VARGRAPHIC is specified:

- If a variable-length field is extracted from an externally described file or an externally described data structure, it is declared as a fixed-length character field.
- For single-byte character fields, the length of the declared field is the length of the DDS field plus 2 bytes.
- For DBCS-graphic data fields, the length of the declared field is twice the length of the DDS field plus 2 bytes.
- The two extra bytes in the field contain a binary number which represents the current length of the variable-length field. Figure 44 on page 146 shows the field length of variable-length fields.
- For variable-length graphic fields defined as fixed-length character fields, the length is double the number of graphic characters.

Single-byte character fields:



Graphic data type fields:

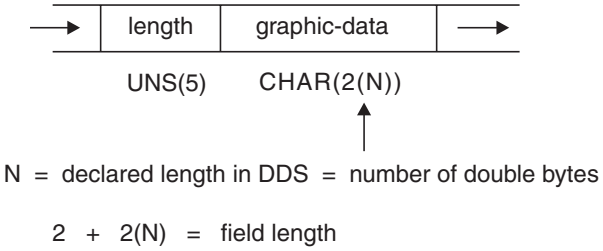


Figure 44. Field Length of Converted Variable Length Fields

- Your program can perform any valid character calculation operations on the declared fixed-length field. However, because of the structure of the field, the first two bytes of the field must contain valid unsigned integer data when the field is written to a file. An I/O exception error occurs for an output operation if the first two bytes of the field contain invalid field length data.
- Field definition conflict errors will occur during a compile when a variable-length field is imported from an OS/400 file into a GUI object and the file is also used as an externally-described file in the program with the *VARCHAR or *VARGRAPHIC option specified. Two bytes for the data length are added to the definition of the field coming from the file record format, which conflicts with the field length definition from the GUI object.

  To circumvent this conflict, do not specify the *VARCHAR or *VARGRAPHIC option, or rename the GUI object and write source code to move data between the two fields as appropriate.
- Field indicators are not allowed on an input specification if the input field is a variable-length field from an externally described input file.
- Keyed operations are not allowed when factor 1 of a keyed operation corresponds to a variable-length key field in an externally described file.
- If you choose to selectively output certain fields in a record and the variable-length field is not specified on the output specification, or if the variable-length field is ignored in the program, a default value is placed in the output buffer of the newly-added record. The default is 0 in the first two bytes and blanks in all of the remaining bytes.
- If you want to change converted variable-length fields, ensure that the current field length is correct. One way to do this is:
  1. Define a data structure with the variable-length field name as a subfield name.
  2. Define a 5-digit unsigned integer subfield overlaying the beginning of the field, and define an N-byte character subfield overlaying the field starting at position 3.
  3. Update the field.

Alternatively, you can move another variable-length field left-aligned into the field. An example of how to change a converted variable-length field in a VARPG program follows.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A*
A*   File MASTER contains a variable-length field
A*
AAN01N02N03T.Name++++++Rlen++TDpBLinPosFunctions++++++++++++++++++++
A*
A          R REC
A            FLDVAR       100          VARLEN
*..1....+....2....+....3....+....4....+....5....+....6....+....7....+.. *
H*
H*  Specify the CVTOPT(*VARCHAR) keyword on a control
H*  specification or compile the VisualAge RPG program with
H*  CVTOPT(*VARCHAR) on the command.
H*
HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H*
H CVTOPT(*VARCHAR)
F*
F*   Externally described file name is MASTER.
F*
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++++
F*
FMASTER    UF   E             DISK
```

Figure 45. Converting a Variable-Length Field in a Program (Part 1 of 2)

```
D*
D*   FLDVAR is a variable-length field defined in DDS with
D*   a DDS length of 100.  Notice that the VARPG field length
D*   is 102.
D*
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D*
D                 DS
D FLDVAR                  1    102
D   FLDLEN                       5U 0 OVERLAY(FLDVAR:1)
D   FLDCHR                     100    OVERLAY(FLDVAR:3)
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C* A character value is moved to the variable length field FLDCHR.
C* After the CHECKR operation, FLDLEN has a value of 5.
C                   READ      MASTER                                 LR
C                   MOVEL     'SALES'       FLDCHR
C     ' '           CHECKR    FLDCHR        FLDLEN
C  NLR              UPDAT     REC
```

Figure 45. Converting a Variable-Length Field in a Program (Part 2 of 2)

If converted variable-length graphic fields are required, you can code a 2-byte unsigned integer field to hold the length, and a graphic subfield of length N to hold the data portion of the field.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
D*
D*   The variable-length graphic field VGRAPH is declared in the
D*   DDS as length 3.  This means the maximum length of the field
D*   is 3 double bytes, or 6 bytes.  The total length of the field,
D*   counting the length portion, is 8 bytes.
D*
D*   Compile the VARPG program with CVTOPT(*VARGRAPHIC).
D*
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D*
D                 DS
DVGRAPH                           8
D  VLEN                           4U 0 OVERLAY(VGRAPH:1)
D  VDATA                          3G   OVERLAY(VGRAPH:3)
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
C*
C*   Assume GRPH is a fixed length graphic field of length 2
C*   double bytes.  Copy GRPH into VGRAPH and set the length of
C*   VGRAPH to 2.
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C                   MOVEL     GRPH          VDATA
C                   Z-ADD     2             VLEN
```

Figure 46. Converting a Variable-Length Graphic Field

Chapter 10. Literals and Named Constants

Literals and named constants are types of constants. Constants can be specified in any of the following places:

- In factor 1
- In factor 2
- In an extended factor 2 on the calculation specifications
- As parameters to keywords on the control specification
- As parameters to built-in functions
- In the Field Name, Constant, or Edit Word fields in the output specifications.
- As array indexes
- With keywords on the definition specification.

Literals

A literal is a self-defining constant that can be referred to in a program. A literal can belong to any of the VisualAge RPG data types.

Character Literals

The following rules apply when specifying a character literal:

- Any combination of characters can be used in a character literal. This includes DBCS characters. DBCS characters must be an even number of bytes. Embedded blanks are valid.
- A character literal with no characters between the apostrophes is allowed.
- Character literals must be enclosed in apostrophes (').
- An apostrophe required as part of a literal is represented by two apostrophes. For example, the literal O'CLOCK is coded as 'O''CLOCK'.
- Character literals are compatible only with character data
- Indicator literals are one byte character literals which contain either '1' (on) or '0' (off).

Hexadecimal Literals

The following rules apply when specifying a hexadecimal literal:

- Hexadecimal literals take the form:

  ```
  X'x1x2...xn'
  ```

  where:

  X'x1x2...xn' must contain the characters A-F, a-f, and 0-9.
- The literal coded between the apostrophes must be of even length.
- Each pair of characters defines a single byte.
- Hexadecimal literals are allowed anywhere that character literals are supported except as factor 2 of ENDSR and as edit words.
- A hexadecimal literal has the same meaning as the corresponding character literal except when used in the bit operations BITON, BITOFF, and TESTB. For the bit operations, factor 2 may contain a hexadecimal literal representing 1 byte. The rules and meaning are the same for hexadecimal literals as for character fields.
- If the hexadecimal literal contains the hexadecimal value for a single quote, it does not have to be specified twice, unlike character literals. For example, the literal `A'B` is specified as `'A''B'` but the hexadecimal version is X'412742' not X'41272742'.

- Normally, hexadecimal literals are compatible only with character data. However, a hexadecimal literal that contains 16 or fewer hexadecimal digits can be treated as an unsigned numeric value when it is used in a numeric expression or when a numeric variable is initialized using the INZ keyword.

Numeric Literals

The following rules apply when specifying a numeric literal:

- A numeric literal consists of any combination of the digits 0 through 9. A decimal point or a sign can be included.
- The sign (+ or –), if present, must be the leftmost character. An unsigned literal is treated as a positive number.
- Blanks cannot appear in a numeric literal.
- Numeric literals are not enclosed in apostrophes (').
- Numeric literals are used in the same way as a numeric field, except that values cannot be assigned to numeric literals.
- The decimal separator may be either a comma or a period

Numeric literals of the float format are specified somewhat differently. Float literals take the form:

```
     <mantissa>E<exponent>
Where
     <mantissa> is a literal as described above with 1 to 16 digits
     <exponent> is a literal with no decimal places, with a value
                between -308 and +308
```

- Float literals do not have to be normalized. That is, the mantissa does not have to be written with exactly one digit to the left of the decimal point. (The decimal point does not even have to be specified.)
- Lower case **e** may be used instead of **E**.
- Either a period ('.') or a comma (',') may be used as the decimal point.
- Float literals are allowed anywhere that numeric constants are allowed except in operations that do not allow float data type. For example, float literals are not allowed in places where a numeric literal with zero decimal positions is expected, such as an array index.
- Float literals follow the same continuation rules as for regular numeric literals. The literal may be split at any point within the literal.
- A float literal must have a value within the limits described in 1.6.2, "Rules for Defining" on page 4.

The following lists some examples of valid float literals:

```
1E1               = 10
1.2e-1            = .12
-1234.9E0         = -1234.9
12e12             = 12000000000000
+67,89E+0003      = 67890  (the comma is the decimal point)
```

The following lists some examples of invalid float literals:

```
1.234E                        <--- no exponent
1.2e-                         <--- no exponent
-1234.9E+309                  <--- exponent too big
12E-2345                      <--- exponent too small
1.797693134862316e308         <--- value too big
179.7693134862316E306         <--- value too big
0.0000000001E-308             <--- value too small
```

Date Literals

Date literals take the form D'xxxxxx' where:

- D indicates that the literal is of type date

- xxxxxx is a valid date in the format specified on the control specification
- xxxxxx is enclosed by apostrophes (').

Time Literals

Time literals take the form T'xxxxxx' where:
- T indicates that the literal is of type time
- xxxxxx is a valid time in the format specified on the control specification
- xxxxxx is enclosed by apostrophes (').

Timestamp Literals

Timestamp literals take the form Z'yyyy-mm-dd-hh.mm.ss.mmmmmm' where:
- Z indicates that the literal is of type timestamp
- yyyy–mm–dd is a valid date (year–month–day)
- hh.mm.ss.mmmmmm is a valid time (hours.minutes.seconds.microseconds)
- yyyy–mm–dd–hh.mm.ss.mmmmmm is enclosed by apostrophes
- Microsecond are optional and if not specified, default to zeros

Graphic Literals

Graphic literals take the form G'K1K2' where:
- G indicates that the literal is of type graphic
- K1K2 is an even number of bytes
- K1K2 is enclosed by apostrophes (').

UCS-2 Literals

UCS-2 literals take the form U'Xxxx...Yyyy' where:
- U indicates that the literal is of type UCS-2.
- Each UCS-2 literal requires four bytes per UCS-2 character in the literal. Each four bytes of the literal represents one double-byte UCS-2 character.
- UCS-2 literals are compatible only with UCS-2 data.

UCS-2 literals are assumed to be in the default UCS-2 CCSID of the module.

Named Constants

A named constant is a symbolic name assigned to a literal. Named constants are defined on definition specifications. The value of a named constant follows the rules specified for literals.

Named Constants

You can give a name to a constant. This name represents a specific value which cannot be changed when the program is running.

Rules for Named Constants

- Named constants can be specified in factor 1, factor 2, and extended-factor 2 on the calculation specifications, as parameters to keywords on the control specification, as parameters to built-in functions, and in the Field Name, Constant, or Edit Word fields in the output specifications. They can also be used as array indexes or with keywords on the definition specification.
- Numeric named constants have no predefined precision. Actual precision is defined by the context that is specified.
- The named constant can be defined anywhere on the definition specifications.

Example of Defining a Named Constant

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
 *  Define a date field and initialize it to the 3rd of September
 *  1988.
 *
D  DateField      S               D   INZ(D'1988-09-03')
 *
 *  Define a binary 9,5 field and initialize it to 0.
 *
D  BIN9_5         S              9B 5 INZ
 *
 *  Define a named constant whose value is the lower case alphabet.
 *
D  Lower          C                   CONST('abcdefghijklmnop-
D                                     qrstuvwxyz')
 *
 *  Define a named constant without explicit use of the keyword CONST.
 *
D  Upper          C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Figure 47. Defining Named Constants

Figurative Constants

The following figurative constants are implied literals that can be specified without a length, because the implied length and decimal positions of a figurative constant are the same as those of the associated field. See "Rules for Figurative Constants" on page 154 for a list of exceptions.

*ALL'x..', *ALLG'K1K2' *ALLU'XxxxYyyy', *ALLX'x1..'	*BLANK/*BLANKS	*HIVAL
*LOVAL *ZERO/*ZEROS	*NULL	*ON/*OFF

Figurative constants can be specified in factor 1 and factor 2 of the calculation specifications. The following shows the reserved words and implied values for figurative constants:

Reserved Words	Implied Values
***BLANK/*BLANKS**	All blanks. Valid only for character, graphic, or UCS-2 fields.
***ZERO/*ZEROS**	Character/numeric fields: All zeros. *For numeric float fields*: The value is '0 E0'.
***HIVAL**	Character, graphic, or UCS-2 fields: The highest collating character for the system (hexadecimal FFs). Numeric fields: All nines with a positive sign. *For Float fields*: *HIVAL for 4-byte float = 3.402 823 5E38 (x'7F7FFFFF') *HIVAL for 8-byte float = 1.797 693 134 862 315 E308 (x'7FEFFFFFFFFFFFFF') Date, time and timestamp fields: See "Date Data" on page 119, "Time Data" on page 135, and "Timestamp Data" on page 137 for *HIVAL values for date, time, and timestamp data.

***LOVAL**	Character, graphic, or UCS-2 fields: The lowest collating character for the system (hexadecimal zeros). Numeric fields: All nines with a negative sign. *For Float fields*: *LOVAL for 4-byte float = -3.402 823 5E38 (x'FF7FFFFF') *LOVAL for 8-byte float = -1.797 693 134 862 315 E308 (x'FFEFFFFFFFFFFFFF') Date, time and timestamp fields: See "Date Data" on page 119, "Time Data" on page 135, and "Timestamp Data" on page 137 for *LOVAL values for date, time, and timestamp data.
***ALL'x..'**	Character/numeric fields: Character string x . . is cyclically repeated to a length equal to the associated field. If the field is a numeric field, all characters within the string must be numeric (0 through 9). No sign or decimal point can be specified when *ALL'x..' is used as a numeric constant. **Note:** You cannot use *ALL'x..' with numeric fields of float format. For numeric integer or unsigned fields, the value is never greater than the maximum value allowed for the corresponding field.
***ALLG'K1K2'**	Graphic fields: The graphic string K1K2 is cyclically repeated to a length equal to the associated field.
***ALLU'XxxxYyyy'**	**UCS-2 fields**: A figurative constant of the form *ALLU'XxxxYyyy' indicates a literal of the form 'XxxxYyyyXxxxYyyy...' with a length determined by the length of the field associated with the *ALLU'XxxxYyyy' constant. Each double-byte character in the constant is represented by four hexadecimal digits. For example, *ALLU'0041' represents a string of repeated UCS-2 'A's.
***ALLX'x1..'**	Character fields: The hexadecimal literal X'x1..' is cyclically repeated to a length equal to the associated field.
***NULL**	A null value valid for basing pointers, procedure pointers, or objects.
***ON/*OFF**	*ON '1' *OFF is '0'. Both are only valid for character fields.

The following figurative constants are implied literals that can be used with the DSPLY operation code:

*ABORT	*CANCEL	*ENTER	*HALT
*IGNORE	*INFO	*NOBUTTON	*OK
*RETRY	*WARN	*YESBUTTON	

The following figurative constants are implied literals that can be used when creating an application's GUI:

*BLACK	*BLUE	*BROWN	*CYAN
*DARKBLUE	*DARKCYAN	*DARKGREEN	*DARKGRAY
*DARKPINK	*DARKRED	*GREEN	*PALEGRAY
*PINK	*RED	*YELLOW	*WHITE

Rules for Figurative Constants

The following rules apply when using figurative constants:

- Figurative constants that are allowed for fixed-length character fields are also allowed for variable-length character fields (*BLANK/*BLANKS, *ZERO/*ZEROS, *HIVAL, *LOVAL, *ALL'x..', *ALLG'K1K2', *ALLX'x1..', *ON/*OFF).

- Figurative constants that are allowed for fixed-length graphic fields are also allowed for variable-length graphic fields (*BLANK/*BLANKS, *HIVAL, *LOVAL, *ALLG'K1K2').
- The figurative constant values are the same for both fixed-length and variable-length character and graphic fields:

```
*HIVAL   = X'FF'
*LOVAL   = X'00'
*BLANK   = ' ' or X'20' or double-byte blank
*ZERO    = '0' or X'30'
*OFF     = '0' or X'30'
*ON      = '1' or X'31'
```

- MOVE and MOVEL operations allow moving a character figurative constant to a numeric field. The figurative constant is first expanded as a zoned numeric with the size of the numeric field, converted to packed or binary numeric if needed, and then stored in the target numeric field. The digit portion of each character in the constant must be valid.
- Figurative constants are considered elementary items. Except for MOVEA, figurative constants act like a field if used in conjunction with an array. For example: MOVE *ALL'XYZ' ARR.

  If ARR has 4-byte character elements, then each element contains 'XYZX'.
- MOVEA is considered to be a special case. The constant is generated with a length equal to the portion of the array specified. For example:
  - MOVEA *BLANK ARR(X)

    Beginning with element X, the remainder of ARR will contain blanks.
  - MOVEA *ALL'XYZ' ARR(X)

    ARR has 4-byte character elements. Element boundaries are ignored, as is always the case with character MOVEA operations. Beginning with element X, the remainder of the array will contain 'XYZXYZXYZ...'.
- The SETGT and SETLL operation codes do not support use of the *HIVAL or *LOVAL value in factor 1.

Note: The results of MOVEA are different from those of the MOVE example:

- After figurative constants are set/reset to their appropriate length, their normal collating sequence can be altered if an alternate collating sequence is specified.
- The move operations MOVE and MOVEL produce the same result when moving the figurative constants *ALL'x..', *ALLG'K1K2', and *ALLX'x1..'. The string is cyclically repeated character by character (starting on the left) until the length of the associated field is the same as the length of the string.
- Figurative constants can be used in compare operations as long as one of the factors is not a figurative constant.
- The figurative constants, *BLANK/*BLANKS, are moved as zeros to a numeric field in a MOVE operation.

Chapter 11. Data Structures

You can define an area in storage and the layout of the fields (subfields) within the area. This area in storage is called a data structure. Specify DS in positions 24 through 25 on a definition specification to define a data structure.

You can use a data structure to:
- Define the same internal area multiple times using different data formats
- Operate on an individual subfield using its name
- Operate on all the subfields as a group using the name of the data structure
- Define a data structure and its subfields in the same way a record is defined
- Define multiple occurrences of a set of data
- Group non-contiguous data into contiguous internal storage locations.

There are three special data structures, each with a specific purpose:
- A data-area data structure (identified by a U in position 23 of the definition specification). See "Position 23 (Type of Data Structure)" on page 260.
- A file information data structure (identified by the keyword INFDS on a file description specifications). See "INFDS(DSname)" on page 248.
- A program-status data structure (identified by an S in position 23 of the definition specification). See "Position 23 (Type of Data Structure)" on page 260.

Data structures can be program-described or externally-described. One data structure can be defined like another using the LIKEDS keyword.

A program-described data structure is identified by a blank in position 22 of the definition specification. The subfield definitions for a program-described data structure must immediately follow the data structure definition. See "Position 22 (External Description)" on page 260.

An externally-described data structure, identified by an E in position 22 of the definition specification, has subfield descriptions contained in an externally-described file. When the program is compiled, the external name is used to locate and extract the external description of the data structure subfields. Specify the name of the external description either in positions 7 through 21, or as a parameter for the keyword EXTNAME. See "Positions 7-21 (Name)" on page 260 and "EXTNAME(file-name{:format-name}{:*ALL| *INPUT|*OUTPUT|*KEY})" on page 272.

Note: The data formats specified for the subfields in the external description are used as the internal formats of the subfields by the compiler. This differs from the way in which externally described files are treated.

An external subfield name can be renamed in the program using the keyword EXTFLD. The keyword PREFIX can be used to add a prefix to the external subfield names that have not been renamed with EXTFLD. Note that the data structure subfields are not affected by the PREFIX keyword specified on a file-description specification even if the file name is the same as the parameter specified in the EXTNAME keyword when defining the data structure using an external file name. Additional subfields can be added to an externally described data structure by specifying program-described subfields immediately after the list of external subfields. See "EXTFLD(field_name)" on page 271 and "PREFIX(prefix{:nbr_of_char_replaced})" on page 293.

Qualifying Data Structure Names

The keyword QUALIFIED indicates that subfields of the data structure are referenced using qualified notation. This permits access by specifying the data structure name followed by a period and the subfield name, for example `DS1.FLD1`. If the QUALIFIED keyword is not used, the subfield name remains unqualified, for example `FLD1`. If QUALIFIED is used the subfield name can be specified by one of the following:

- A "**Simply Qualified Name**" is a name of the form "**A.B**". Simply qualified names are allowed as arguments to keywords on File and Definition Specifications; in the Field-Name entries on Input and Output Specifications; and in the Factor 1, Factor 2, and Result-Field entries on fixed-form calculation specifications, i.e.`dsname.subf`. While spaces are permitted between elements of a fully-qualified name, they are not permitted in simply qualified names.
- A "**Fully Qualified Name**" is a name with qualification and indexing to an arbitrary number of levels, for example, "**A(X).B.C(Z+17)**". Fully qualified names are allowed in any free-form calculation specifications, or in any Extended-Factor-2 entry. This includes operations codes CLEAR and DSPLY coded in free-form calculations.

In addition, arbitrary levels of indexing and qualification are allowed. For example, a programmer could code:`ds(x).subf1.s2.s3(y+1).s4` as an operand within an expression. Please see "QUALIFIED" on page 293 for further information on the use of the QUALIFIED keyword.

Fully qualified names may be specified as the Result-Field operand for opcodes CLEAR and DSPLY when coded in free-form calc specs. An expression is allowed for the Factor 1 operand for opcode DSPLY (coded in free-form calculation specifications), however, if the operand is more complex than a fully qualified name, the expression must be enclosed in parentheses.

Array Data Structures

An "**Array Data Structure**" is a data structure defined with keyword DIM. An array data structure is like a multiple-occurrence data structure, except that the index is explicitly specified, as with arrays.

Notes:

1. Keyword DIM is allowed for data structures defined as QUALIFIED.
2. When keyword DIM is coded for a data structure or LIKEDS subfield, array keywords CTDATA, FROMFILE, and TOFILE are not allowed. In addition, the following data structure keywords are not allowed for an array data structure:
   - DTAARA
   - OCCURS.
3. For a data structure X defined with LIKEDS(Y), if data structure Y is defined with keyword DIM, data structure X is not defined as an array data structure.
4. If X is a subfield in array data structure DS, then an array index must be specified when referring to X in a qualified name. In addition, the array index may not be *. Within a fully qualified name expression, an array index may only be omitted (or * specified) for the right-most name.

Defining Data Structure Parameters in a Prototype or Procedure Interface

To define a prototyped parameter as a data structure, you must first define the layout of the parameter by defining an ordinary data structure. Then, you can define a prototyped parameter as a data structure by using the LIKEDS keyword. To use the subfields of the parameter, specify the subfields qualified with parameter name: dsparm.subfield. For example

```
 * PartInfo is a data structure describing a part.
D  PartInfo       DS                  QUALIFIED
D  Manufactr                     4
D  Drug                          6
D  Strength                      3
D  Count                         3  0
 * Procedure "Proc" has a parameter "Part" that is a data
 * structure whose subfields are the same as the subfields
 * in "PartInfo".  When calling this procedure, it is best
 * to pass a parameter that is also defined LIKEDS(PartInfo)
 * (or pass "PartInfo" itself), but the compiler will allow
 * you to pass any character field that has the correct
 * length.
D Proc            PR
D  Part                          LIKEDS(PartInfo)
P Proc            B
 * The procedure interface also defines the parameter Part
 * with keyword LIKEDS(PartInfo).
 * This means the parameter is a data structure, and the subfields
 * can be used by specifying them qualified with "Part.", for
 * example "Part.Strength"
D Proc            PI
D  Part                          LIKEDS(PartInfo)
C                   IF        Part.Strength > getMaxStrength (Part.Drug)
C                   CALLP     PartError (Part : DRUG_STRENGTH_ERROR)
C                   ELSE
C                   EVAL      Part.Count = Part.Count + 1
C                   ENDIF
P Proc            E
```

Defining Data Structure Subfields

You define a subfield by specifying blanks in the Definition-Type entry (positions 24 through 25) of a definition specification. The subfield definition(s) must immediately follow the data structure definition. The subfield definitions end when a definition specification with a non-blank Definition-Type entry is encountered, or when a different specification type is encountered.

The name of the subfield is entered in positions 7 through 21. To improve readability of your source, you may want to indent the subfield names to show visually that they are subfields.

If the data structure is defined with the QUALIFIED keyword, the subfield names can be the same as other names within your program. The subfield names will be qualified by the owning data structure when they are used.

You can also define a subfield like an existing item using the LIKE keyword. When defined in this way, the subfield receives the length and data type of the item on

which it is based. Similarly, you can use the LIKEDS keyword to define an entire data structure like an existing item. See Figure 92 on page 279 for an example using the LIKE keyword.

The keyword LIKEDS is allowed on any subfield definition. When specified, the subfield is defined to be a data structure, with its own set of subfields. If data structure DS has subfield S1 which is defined like a data structure with a subfield S2, a programmer must refer to S2 using the expression `DS.S1.S2.`

Notes:

1. Keyword LIKEDS is allowed for subfields only within QUALIFIED data structures.
2. Keywords DIM and LIKEDS are both allowed on the same subfield definition.

You can overlay the storage of a previously defined subfield with that of another subfield using the OVERLAY keyword. The keyword is specified on the later subfield definition.

Specifying Subfield Length

The length of a subfield may be specified using absolute (positional) or length notation, or its length may be implied.

Absolute
Specify a value in both the From-Position (positions 26 through 32) and the To-Position/Length (positions 33 through 39) entries on the definition specification.

Length
Specify a value in the To-Position/Length (positions 33 through 39) entry. The From-Position entry is blank.

Implied Length
If a subfield appears in the first parameter of one or more OVERLAY keywords, the subfield can be defined without specifying any type or length information. In this case, the type is character and the length is determined by the overlaid subfields.

In addition, some data types, such as Pointers, Dates, Times and Timestamps have a fixed length. For these types, the length is implied, although it can be specified.

When using length notation, the subfield is positioned such that its starting position is greater than the maximum To-Position of all previously-defined subfields. For examples of each notation, see "Data Structure Examples" on page 162.

Aligning Data Structure Subfields

Alignment of subfields may be necessary. In some cases it is done automatically; in others, it must be done manually.

For example, when defining subfields of type basing pointer or procedure pointer using the length notation, the compiler will automatically perform padding if necessary to ensure that the subfield is aligned properly.

When defining float, integer or unsigned subfields, alignment may be desired to improve runtime performance. If the subfields are defined using length notation,

you can automatically align float, integer or unsigned subfields by specifying the keyword ALIGN on the data structure definition. However, note the following exceptions:

- The ALIGN keyword is not allowed for a file information data structure or a program status data structure.
- Subfields defined using the keyword OVERLAY are not aligned automatically, even if the keyword ALIGN is specified for the data structure. In this case, you must align the subfields manually.

Automatic alignment will align the fields on the following boundaries.

- 2 bytes for 5-digit integer or unsigned subfields
- 4 bytes for 10-digit integer or unsigned subfields, or 4-byte float subfields
- 8 bytes for 20-digit integer or unsigned subfields
- 8 bytes for 8-byte float subfields
- 16 bytes for pointer subfields

If you are aligning fields manually, make sure that they are aligned on the same boundaries. A start-position is on an n-byte boundary if `((position - 1) mod n) = 0`. (The value of ″x mod y″ is the remainder after dividing x by y in integer arithmetic. It is the same as the MVR value after X DIV Y.)

Figure 48 shows a sequence of bytes and identifies the different boundaries used for alignment.

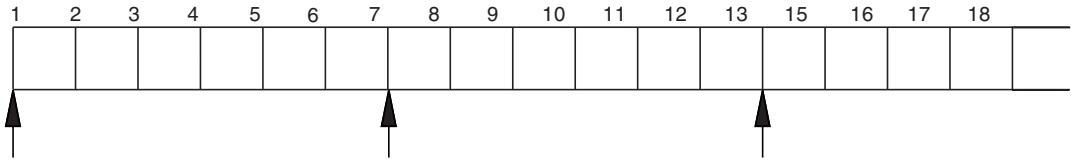


Figure 48. Boundaries for Data Alignment

Note the following about the preceding byte sequence:

- Position 1 is on a 16-byte boundary, since ((1-1) mod 16) = 0.
- Position 13 is on a 4-byte boundary, since ((13-1) mod 4) = 0.
- Position 7 is *not* on a 4-byte boundary, since ((7-1) mod 4) = 2.

Initialization of Nested Data Structures

The keyword INZ(*LIKEDS) is allowed on a LIKEDS subfield. The LIKEDS subfield is initialized exactly the same as the corresponding data structure.

Keyword INZ is allowed on a LIKEDS subfield. All nested subfields of the LIKEDS subfield are initialized to their default values. This also applies to more deeply nested LIKEDS subfields, with the exception of nested LIKEDS subfields with INZ(*LIKEDS) specified.

If keyword INZ is coded on a main data structure definition, keyword INZ is implied on all subfields of the data structure without explicit initialization. This includes LIKEDS subfields.

Special Data Structures

Special data structures include:
- Data area data structures
- File information data structures (INFDS)
- Program-status data structures

For examples, see "Data Structure Examples"

Note that the above data structures cannot be defined in subprocedures.

Data-Area Data Structure

A data-area data structure is specified by a U in position 23 of the definition specification. This indicates that the same data area that is read and locked at program initialization should be written out and unlocked at the end of the program. Data-area data structures, like other data structures, have the type character. A data area read into a data area data structure must also be character. The data area and data-area data structure must have the same name unless you rename the data area in the program by using the *DTAARA DEFINE operation code or the DTAARA keyword. See "DEFINE (Field Definition)" on page 548 and "DTAARA{(*VAR:)data_area_name}" on page 270.

You can specify the data area operations (IN, OUT, and UNLOCK) for a data area that is implicitly read in and written out. Before you use a data area data structure with these operations, you must specify that data area in the result field of the *DTAARA DEFINE operation or with the DTAARA keyword. See "DEFINE (Field Definition)" on page 548 and "DTAARA{(*VAR:)data_area_name}" on page 270.

Note: A data-area data structure cannot be specified in the result field of a PARM operation in the *ENTRY PLIST.

File Information Data Structure

You can specify a file information data structure for each file in the program. File information data structures are defined by the keyword INFDS on a file description specifications. See "INFDS(DSname)" on page 248. This provides you with status information on the file exception or error that occurred. The file information data structure name must be unique for each file. A file information data structure contains subfields that provide information on the file exception or error that occurred. For more information on file information data structures and their subfields, see "File Information Data Structure" on page 41.

Program-Status Data Structure

A program-status data structure provides program exception and error information to the program. It identified by an S in position 23 of the definition specification. For more information on program-status data structures and their subfields, see "Program Status Data Structure" on page 51.

Data Structure Examples

The following examples show various uses for data structures and how to define them.

Example	Description
Figure 49 on page 163	Using a data structure to subdivide a field

Example	Description
Figure 50 on page 164	Using a data structure to group fields
Figure 51 on page 165	Using keywords QUALIFIED, LIKEDS, and DIM with data structures, and how to code fully-qualified subfields
Figure 52 on page 166	Data structure with absolute and length notation
Figure 53 on page 166	Rename and initialize an externally described data structure
Figure 54 on page 167	Using PREFIX to rename all fields in an external data structure
Figure 55 on page 167	Defining a multiple occurrence data structure
Figure 56 on page 168	Aligning data structure subfields
Figure 57 on page 169	Using data area data structures

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.....................................Keywords++++++++++++++++++++++++++++++
 *
 * Use length notation to define the data structure subfields.
 * You can refer to the entire data structure by using Partno, or by
 * using the individual subfields Manufactr, Drug, Strength or Count.
 *
D Partno          DS
D  Manufactr                     4
D  Drug                          6
D  Strength                      3
D  Count                         3  0
D
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC..............................
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr......
 *
 *  Records in program described file FILEIN contain a field, Partno,
 *  which needs to be subdivided for processing in this program.
 *  To achieve this, the field Partno is described as a data structure
 *  using the above Definition specification
 *
IFILEIN    NS  01   1 CA    2 CB
I                                3   18  Partno
I                               19   29  Name
I                               30   40  Patno
```

Figure 49. Using a Data Structure to Subdivide a Field

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.....................................Keywords++++++++++++++++++++++++++++++
 *
 *  When you use a data structure to group fields, fields from
 *  non-adjacent locations on the input record can be made to occupy
 *  adjacent internal locations. The area can then be referred to by
 *  the data structure name or individual subfield name.
 *
D Partkey         DS
D  Location                      4
D  Partno                        8
D  Type                          4
D
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC.................................
I.......................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr......
 *
 *  Fields from program described file TRANSACTN need to be
 *  compared to the field retrieved from an Item_Master file
 *
ITRANSACTN NS  01   1 C1    2 C2
I                                    3   10  Partno
I                                   11   16 0Quantity
I                                   17   20  Type
I                                   21   21  Code
I                                   22   25  Location
I
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  Use the data structure name Partkey, to compare to the field
 *  Item_Nbr
 *
C                   :
C     Partkey       IFEQ      Item_Nbr                              99
C                   :
C*
```

Figure 50. Using a Data Structure to Group Fields

```
D CustomerInfo     DS                        QUALIFIED BASED(@)
D    Name                              20A
D    Address                           50A

D ProductInfo      DS                        QUALIFIED BASED(@)
D    Number                             5A
D    Description                       20A
D    Cost                               9P 2

D SalesTransaction...
D                  DS                        QUALIFIED
D    Buyer                                   LIKEDS(CustomerInfo)
D    Seller                                  LIKEDS(CustomerInfo)
D    NumProducts                       10I 0
D    Products                                LIKEDS(ProductInfo)
D                                            DIM(10)

 /free
     TotalCost = 0;
     for i = 1 to SalesTransation. Numproducts;
         TotalCost = TotalCost + SalesTransaction.Products (i).Cost;
         dsply SalesTransaction.Products (i).Cost;
     endfor;
     dsply ('Total cost is ' + %char(TotalCost));
 /end-free
```

Figure 51. Using Keywords QUALIFIED, LIKEDS and DIM with data structures

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
 *  Define a program described data structure called FRED
 *  The data structure is composed of 5 fields:
 *   1.   An array with element length 10 and dimension 70(Field1)
 *   2.   A field of length 30 (Field2)
 *   3/4. Divide Field2 in 2 equal length fields (Field3 and Field4)
 *   5.   Define a binary field over the 3rd field
 *  Note the indentation to improve readability
 *
 *
 *  Absolute notation:
 *
 *  The compiler will determine the array element length (Field1)
 *  by dividing the total length (700) by the dimension (70)
 *
D FRED            DS
D  Field1                  1    700    DIM(70)
D  Field2                701    730
D   Field3               701    715
D    Field5              701    704B 2
D   Field4               716    730
 *
 *  Length notation:
 *
 *  The OVERLAY keyword is used to subdivide Field2
 *
D FRED            DS
D  Field1                       10    DIM(70)
D  Field2                       30
D   Field3                      15    OVERLAY(Field2)
D    Field5                      4B 2 OVERLAY(Field3)
D   Field4                      15    OVERLAY(Field2:16)
```

Figure 52. Data Structure with Absolute and Length Notation

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
 *  Define an externally described data structure with internal name
 *  FRED and external name EXTDS and rename field CUST to CUSTNAME
 *  Initialize CUSTNAME to 'GEORGE' and PRICE to 1234.89.
 *  Assign to subfield ITMARR (defined in the external description as a
 *  100 byte character field) the DIM keyword
 *
D Fred          E DS                  EXTNAME(EXTDS)
D   CUSTNAME    E                     EXTFLD(CUST) INZ('GEORGE')
D   PRICE       E                     INZ(1234.89)
D   ITMARR      E                     DIM(10)
```

Figure 53. Rename and Initialize an Externally Described Data Structure

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
D
D extds1        E DS                  EXTNAME (CUSTDATA)
D                                     PREFIX (CU_)
D   Name        E                     INZ ('Joe's Garage')
D   Custnum     E                     EXTFLD (NUMBER)
D
 *
 *  The previous data structure will expand as follows:
 *  -- All externally described fields are included in the data
 *     structure
 *  -- Renamed subfields keep their new names
 *  -- Subfields that are not renamed are prefixed with the
 *     prefix string
 *
 *  Expanded data structure:
 *
D EXTDS1        E DS
D   CU_NAME     E               20A   EXTFLD (NAME)
D                                     INZ ('Joe's Garage')
D   CU_ADDR     E               50A   EXTFLD (ADDR)
D   CUSTNUM     E                9S0  EXTFLD (NUMBER)
D   CU_SALESMN  E                7P0  EXTFLD (SALESMN)
```

Figure 54. Using PREFIX to Rename All Fields in an External Data Structure

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
 *  Define a Multiple Occurrence data structure of 20 elements with:
 *  -- 3 fields of character 20
 *  -- A 4th field of character 10 which overlaps the 2nd
 *     field starting at the second position.
 *
 *  Named constant 'twenty' is used to define the occurrence
 *
 *  Absolute notation (using begin/end positions)
 *
D twenty          C                   CONST(20)
D
DDataStruct       DS                  OCCURS (twenty)
D field1                   1     20
D field2                  21     40
D  field21                22     31
D field3                  41     60
 *
 *  Mixture of absolute and length notation
 *
D DataStruct      DS                  OCCURS(twenty)
D  field1                         20
D  field2                         20
D   field21               22     31
D  field3                 41     60
```

Figure 55. Defining a Multiple Occurrence Data Structure

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 * Data structure with alignment:
D MyDS            DS                  ALIGN
 *  Properly aligned subfields
 *    Integer subfields using absolute notation.
D    Subf1                33    34I 0
D    Subf2                37    40I 0
 *    Integer subfields using length notation.
 *    Note that Subf3 will go directly after Subf2
 *    since positions 41-42 are on a 2-byte boundary.
 *    However, Subf4 must be placed in positions 45-48
 *    which is the next 4-byte boundary after 42.
D    Subf3                      5I 0
D    Subf4                     10I 0
 *    Integer subfields using OVERLAY.
D    Group              101   120A
D      Subf6                    5I 0 OVERLAY (Group: 3)
D      Subf7                   10I 0 OVERLAY (Group: 5)
D      Subf8                    5U 0 OVERLAY (Group: 9)
 *  Subfields that are not properly aligned:
 *    Integer subfields using absolute notation:
D    SubfX1               10    11I 0
D    SubfX2               15    18I 0
 *    Integer subfields using OVERLAY:
D    BadGroup           101   120A
D      SubfX3                   5I 0 OVERLAY (BadGroup: 2)
D      SubfX4                  10I 0 OVERLAY (BadGroup: 6)
D      SubfX5                  10U 0 OVERLAY (BadGroup: 11)
 *    Integer subfields using OVERLAY:
D    WorseGroup         200   299A
D      SubfX6                   5I 0 OVERLAY (WorseGroup)
D      SubfX7                  10I 0 OVERLAY (WorseGroup: 3)
 *
 * The subfields receive warning messages for the following reasons:
 * SubfX1 - end position (11) is not a multiple of 2 for a 2 byte field.
 * SubfX2 - end position (18) is not a multiple of 4 for a 4 byte field.
 * SubfX3 - end position (103) is not a multiple of 2.
 * SubfX4 - end position (109) is not a multiple of 4.
 * SubfX5 - end position (114) is not a multiple of 4.
 * SubfX6 - end position (201) is not a multiple of 2.
 * SubfX7 - end position (205) is not a multiple of 4.
```

Figure 56. Aligning Data Structure Subfields

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
 *
 *  This program uses a data-area data structure to accumulate
 *  a series of totals.
 *
D  Totals        UDS
D    Tot_amount                  8 2
D    Tot_gross                  10 2
D    Tot_netto                  10 2
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CSRN01Factor1+++++++Opcode(E)+Factor2++++++++++++++++++++++++++++++++++++++++
 *
C                   :
C                   EVAL      Tot_amount = Tot_amount + amount
C                   EVAL      Tot_gross  = Tot_gross  + gross
C                   EVAL      Tot_netto  = Tot_netto  + netto
```

Figure 57. Using Data-area Data Structures

Chapter 12. Using Arrays and Tables

Arrays and tables are both collections of data fields (elements) of the same:
- Field length
- Data type
  - Character
  - Numeric
  - Data Structure
  - Date
  - Time
  - Timestamp
  - Graphic
  - Basing Pointer
  - Procedure Pointer
  - UCS-2
- Format
- Number of decimal positions (if numeric)

Arrays and tables differ in that:
- You can refer to a specific array element by its position
- You cannot refer to specific table elements by their position
- An array name by itself refers to all elements in the array
- A table name always refers to the element found in the last LOOKUP (Look Up a Table or Array Element) operation. .

Note: You can define only run-time arrays in a subprocedure. Tables, pre-runtime arrays, and compile-time arrays are not supported. If you want to use a pre-run array or compile-time array in a subprocedure, you must define it in the main source section.

The following sections describe how to use arrays:
- "Arrays"
- "Initializing Arrays" on page 178
- "Defining Related Arrays" on page 178
- "Searching Arrays" on page 180
- "Using Arrays" on page 183
- "Array Output" on page 184

"Tables" on page 186 describes the same information for tables.

"Arrays" describes how to code an array, how to specify the initial values of the array elements, how to change the values of an array, and the special considerations for using an array.

Arrays

There are three types of arrays:
- The runtime array is loaded while the program is running.
- The compile-time array is loaded when your program is created. The initial data becomes a permanent part of your program.
- The pre-runtime array is loaded from an array file when your program begins running, before any input, calculation, or output operations are processed.

The essentials of defining and loading an array are described for a runtime array. For defining and loading compile-time and pre-runtime arrays, use these essentials and some additional specifications.

Array Name and Index

You refer to an entire array using the array name alone. You refer to the individual elements of an array using the array name, followed by a left parenthesis, followed by an index, followed by a right parenthesis. For example:

```
AR(IND)
```

The index indicates the position of the element within the array (starting from 1) and is either a number or a field containing a number.

The following rules apply when specifying an array name and index:

- The array name must be a unique symbolic name
- The index must be a numeric field or constant greater than zero and with zero decimal positions
- If the array is specified within an expression in the extended factor 2 field, the index may be an expression returning a numeric value with zero decimal positions
- At run time, if the program refers to an array using an index with a value that is zero, negative, or greater than the number of elements in the array, then the error/exception routine takes control of the program.

Essential Array Specifications

You define an array on a definition specification:

- Specify the array name in positions 7 through 21
- Specify the number of entries in the array using the DIM keyword
- Specify length, data format, and decimal positions as you would any scalar fields. You may specify explicit From- and To-position entries (if defining a subfield), or an explicit Length-entry; or you may define the array attributes using the LIKE keyword; or the attributes may be specified elsewhere in the program.
- If you need to specify a sort sequence, use the ASCEND or DESCEND keywords.

Figure 58 shows an example of the essential array specifications.

Coding a Runtime Array

If you make no further specifications beyond the essential array specifications, you have defined a runtime array. Note that the keywords ALT, CTDATA, EXTFMT, FROMFILE, PERRCD, and TOFILE cannot be used for a runtime array.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
DARC              S              3A   DIM(12)
```

Figure 58. The Essential Array Specifications to Define a Runtime Array

Loading a Runtime Array

You can assign initial values for a runtime array using the INZ keyword on the definition specification. You can also assign initial values for a runtime array through input or calculation specifications. This second method can also be used to put data into other types of arrays.

For example, you can use the calculation specifications for the MOVE operation to put 0 in each element of an array (or in selected elements).

Using the input specifications, you can fill an array with the data from a file. The following sections provide more details on retrieving this data from the records of a file.

Note: Date and time runtime data must be in the same format and use the same separators as the date or time array being loaded.

Loading a Runtime Array in One Source Record

If the array information is contained in one record, the information can occupy consecutive positions in the record or it can be scattered throughout the record.

If the array elements are consecutive on the input record, the array can be loaded with a single input specification. Figure 59 shows the specifications for loading an array of six elements (12 characters each) from a single record.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
DINPARR           S             12A   DIM(6)
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC..............................
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
IARRFILE   AA  01
I                                1   72  INPARR
```

Figure 59. Using a Runtime Array with Consecutive Elements

If the array elements are scattered throughout the record, they can be defined and loaded one at a time, with one element described on a specification line.

Figure 60 shows the specifications for loading an array of six elements (12 characters each) from a single record. A blank separates each of the elements from the others.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
DARRX             S             12A   DIM(6)
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC..............................
I.....................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr....
IARRFILE   AA  01
I                                 1   12  ARRX(1)
I                                14   25  ARRX(2)
I                                27   38  ARRX(3)
I                                40   51  ARRX(4)
I                                53   64  ARRX(5)
I                                66   77  ARRX(6)
```

Figure 60. Defining a Runtime Array with Scattered Elements

Loading a Runtime Array Using Multiple Source Records

If the array information is in more than one record, you can use various methods to load the array. The method to use depends on the size of the array and whether or not the array elements are consecutive in the input records. Records are processed one record at a time. Therefore the entire array is not processed until all the records containing the array information are read and the information is moved into the array fields. It may be necessary to suppress calculation and output operations until the entire array is read into the program.

Sequencing Runtime Arrays

Runtime arrays are not sequence checked. If you process a SORTA (sort an array) operation, the array is sorted into the sequence specified on the definition specification (the ASCEND or DESCEND keywords) defining the array. If the sequence is not specified, the array is sorted into ascending sequence. When the high (positions 71 and 72 of the calculation specifications) or low (positions 73 and 74 of the calculation specifications) indicators are used in the LOOKUP operation, the array sequence must be specified.

Coding a Compile-Time Array

A compile-time array is specified using the essential array specifications and the keyword CTDATA. You can specify the number of array entries in an input record using the PERRCD keyword on the definition specification. If you do not specify the PERRCD keyword, the number of entries defaults to 1. See the specifications in Figure 61 on page 175 for an example.

You can specify the external data format using the EXTFMT(code) keyword. See "EXTFMT(code)" on page 271 for more information.

Note: The EXTFMT keyword cannot be used if the array data resides on the workstation. The EXTFMT keyword is not allowed for float compile-time arrays.

The TOFILE keyword can be used to specify a file to which the array is to be written when the program ends with LR on.

Loading a Compile-Time Array

For a compile-time array, enter array source data into records in the program source member. If you use the **CTDATA keyword, the array data may be entered in anywhere following the source records. If you do not use this keyword, the array data must follow the source records in the order in which the compile-time arrays and tables were defined on the definition specifications. This data is loaded into the array when the program is compiled. Until the program is recompiled with new data, the array will always initially have the same values each time you call the program unless the previous call ended with LR off.

Compile-time arrays can be described separately or in alternating format (with the ALT keyword). Alternating format means that the elements of one array are intermixed on the input record with elements of another array.

Rules for Array Source Records

The rules for array source records are:

- The first array entry for each record must begin in position 1.
- All elements must be the same length and follow each other with no intervening spaces
- An entire record need not be filled with entries. If it is not, blanks or comments can be included after the entries. See Figure 61 on page 175.
- If the number of elements in the array as specified on the definition specification is greater than the number of entries provided, the remaining elements are filled with the default values for the data type specified.

```
     DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++
     DARC              S              3A   DIM(12) PERRCD(5) CTDATA
**CTDATA ARC
48K16343J64044HComments can be placed here
12648A47349K346Comments can be placed here
50B125         Comments can be placed here
```

48K	163	43J	640	44H	126	48A	473	49K	346	50B	125

This is the compile-time array, ARC.

Figure 61. Array Source Record with Comments

- Each record, except the last, must contain the number of entries specified with the PERRCD keyword on the definition specifications. In the last record, unused entries must be blank and comments can be included after the unused entries.
- Each entry must be contained entirely on one record. An entry cannot be split between two records. The length of a single entry is limited to the maximum length of 100 characters (size of source record). If arrays are used and are described in alternating format, corresponding elements must be on the same record. Together they cannot exceed 100 characters.
- For date and time compile-time arrays the data must be in the same format and use the same separators as the date or time array being loaded.
- Array data may be specified in one of two ways:
  - **CTDATA arrayname: The data for the array may be specified anywhere in the compile-time data section.
  - **b: (b=blank) The data for the arrays must be specified in the same order in which they are specified in the Definition specifications.

  Only one of these techniques may be used in one program.
- Arrays can be in ascending(ASCEND keyword), descending (DESCEND keyword), or no sequence (no keyword specified).
- Graphic and UCS-2 arrays are sorted by hexadecimal values.
- If L or R is specified on the EXTFMT keyword on the definition specification, each element must include the sign (+ or −). For example, an array with an element size of 2 with L specified would require 3 positions in the source data (+37−38+52−63).
- Float compile-time data are specified in the source records as float or numeric literals. Arrays defined as 4-byte float require 14 positions for each element; arrays defined as 8-byte float require 23 positions for each element.

Coding a Pre-Runtime Array

On the definition specifications, in addition to the essential array specifications, you can specify the name of the file with the array input data, using the FROMFILE keyword. You can use the TOFILE keyword to specify the name of a file to which the array is written at the end of the program. If the file is a combined file (specified by a C in position 17 of the file description specifications), the parameter for the FROMFILE and TOFILE keywords must be the same. You can use the PERRCD keyword to specify the number of elements per input record.

On the EXTFMT keyword, specify:
- B if the data is in binary format
- L to indicate a sign on the left of a data element
- P if the array data is in packed decimal format
- R to indicate a sign on the right of a data element
- S if the array data is in zoned decimal format.

Specify a T in position 18 of the file description specifications for the file with the array input data.

To compare the coding of two pre-runtime arrays, a compile-time array, and a runtime array, see Figure 62 on page 177.

The ALT keyword can be used to specify arrays in alternating format. (See Figure 62 on page 177.)

Note: The integer or unsigned format cannot be specified for arrays defined with more than ten digits.

```
*....+....1....+....2....+....3....+....4....+....5....+....6....+....*
      HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      H DATFMT(*USA) TIMFMT(*HMS)
      DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++
      D* Runtime array.  ARI has 10 elements of type date. They are
      D* initialized to September 15, 1994.  This is in month, day,
      D* year format using a slash as a separator as defined on the
      D* control specification.
      DARI              S               D   DIM(10) INZ(D'09/15/1994')
      D* Compile-time arrays in alternating format.  Both arrays have
      D* eight elements (three elements per record).  ARC is a character
      D* array of length 15, and ARD is a time array with a predefined
      D* length of 8.
      DARC              S             15    DIM(8) PERRCD(3)
      D                                     CTDATA
      DARD              S               T   DIM(8) ALT(ARC)
      D*
      D* Pre-runtime array.  ARE, which is to be read from file DISKIN,
      D* has 250 character elements (12 elements per record).  Each
      D* element is five positions long.  The size of each record
      D* is 60 (5*12). The elements are arranged in ascending sequence.
      DARE              S              5A   DIM(250) PERRCD(12) ASCEND
      D                                     FROMFILE(DISKIN)
      D*
      D*
      D* Pre-runtime array specified as a combined file.  ARH is written
      D* back to the same file from which it is read when the program
      D* ends normally with LR on.  ARH has 250  character elements
      D* (12 elements per record).  Each elements is five positions long.
      D* The elements are arranged in ascending sequence.
      DARH              S              5A   DIM(250) PERRCD(12) ASCEND
      D                                     FROMFILE(DISKOUT)
      D                                     TOFILE(DISKOUT)
**CTDATA ARC
Toronto        12:15:00Winnipeg       13:23:00Calgary        15:44:00
Sydney         17:24:30Edmonton       21:33:00Saskatoon      08:40:00
Regina         12:33:00Vancouver      13:20:00
```

Figure 62. Definition Specifications for Different Types of Arrays

Loading a Pre-Runtime Array

For a pre-runtime array, enter array input data into a sequential program-described file. When you call a program, but before any input, calculation, or output operations are processed, the array is loaded with initial values from the file. By modifying this file, you can alter the array's initial values on the next call to the program, without recompiling the program. The file is read in arrival sequence. The rules for pre-runtime array data are the same as for compile-time array data, except there are no restrictions on the length of each record. See "Rules for Array Source Records" on page 174.

Sequence Checking for Character Arrays

When sequence checking for character arrays occurs, VisualAge RPG uses the default ASCII collating sequence.

Initializing Arrays

To initialize each element in a runtime array to the same value, specify the INZ keyword on the definition specification. If the array is defined as a data structure subfield, the normal rules for data structure initialization overlap apply (the initialization is done in the order that the fields are declared within the data structure).

Compile-Time and Pre-Runtime Arrays

The INZ keyword cannot be specified for a compile-time or pre-runtime array, because their initial values are assigned to them through other means (compile-time data or data from an input file). If a compile-time or pre-runtime array appears in a globally initialized data structure, it is not included in the global initialization.

Note: Compile-time arrays are initialized in the order in which the data is declared after the program, and pre-runtime arrays are initialized in the order of declaration of their initialization files, regardless of the order in which these arrays are declared in the data structure. Pre-runtime arrays are initialized after compile-time arrays.

If a subfield initialization overlaps a compile-time or pre-runtime array, the array is initialized after the subfield, regardless of the order in which fields are declared within the data structure.

Defining Related Arrays

You can load two compile-time arrays or two pre-runtime arrays in alternating format by using the ALT keyword on the definition of the alternating array. You specify the name of the primary array as the parameter for the ALT keyword. The records for storing the data for such arrays have the first element of the first array followed by the first element of the second array, the second element of the first array followed by the second element of the second array, the third element of the first array followed by the third element of the second array, and so on. Corresponding elements must appear on the same record. The PERRCD keyword on the main array definition specifies the number of corresponding pairs per record, each pair of elements counting as a single entry. You can specify EXTFMT on both the main and alternating array.

Figure 63 on page 179 shows two arrays in alternating format.

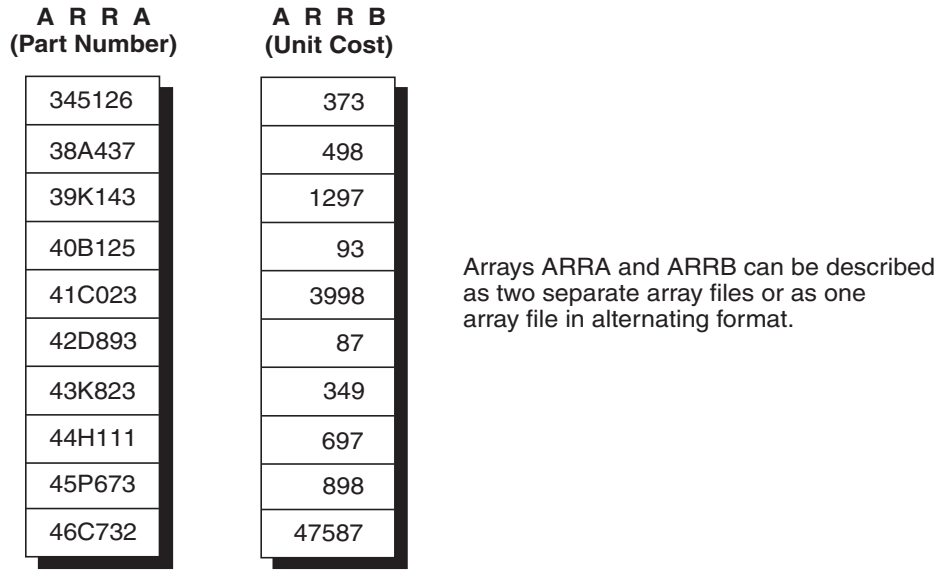


Figure 63. Arrays in Alternating and Nonalternating Format

The records for ARRA and ARRB look like the records in Figure 64 when described as two separate array files.

This record contains ARRA entries in positions 1 through 60.

ARRA entry 1	ARRA entry 7	ARRA entry 13	ARRA entry 19	ARRA entry 25	ARRA entry 31	ARRA entry 37	ARRA entry 43	ARRA entry 49	ARRA entry 55 . . . .

Figure 64. Arrays Records for Two Separate Array Files

This record contains ARRB entries in positions 1 through 50.

ARRB entry 1	ARRB entry 6	ARRB entry 11	ARRB entry 16	ARRB entry 21	ARRB entry 26	ARRB entry 31	ARRB entry 36	ARRB entry 41	ARRB entry 46 . . . .

Figure 65. Arrays Records for One Array File

The records for ARRA and ARRB look like the records below in Figure 66 when described as one array file in alternating format. The first record contains ARRA and ARRB entries in alternating format in positions 1 through 55. The second record contains ARRA and ARRB entries in alternating format in positions 1 through 55.

ARRA entry 1	ARRB entry 1	ARRA entry 7	ARRB entry 6	ARRA entry 13	ARRB entry 11	ARRA entry 19	ARRB entry 16	ARRA entry 25	ARRB entry 21 . . . .

Figure 66. Array Records for One Array File in Alternating Format

```
*....+....1....+....2....+....3....+....4....+....5....+....6....+....*
      DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++
      DARRA             S              6A   DIM(6) PERRCD(1) CTDATA
      DARRB             S              5  0 DIM(6) ALT(ARRA)
      DARRGRAPHIC       S              3G   DIM(2) PERRCD(2) CTDATA
      DARRC             S              3A   DIM(2) ALT(ARRGRAPHIC)
      DARRGRAPH1        S              3G   DIM(2) PERRCD(2) CTDATA
      DARRGRAPH2        S              3G   DIM(2) ALT(ARRGRAPH1)
**CTDATA ARRA
345126  373
38A437  498
39K143 1297
40B125   93
41C023 3998
42D893   87
**CTDATA ARRGRAPHIC
ok1k2k3iabcok4k5k6iabc
**CTDATA ARRGRAPH1
ok1k2k3k4k5k6k1k2k3k4k5k6i
```

Figure 67. Arrays Records for One Array File in Alternating Format

Searching Arrays

The following can be used to search arrays:
- The LOOKUP operation code
- The %LOOKUP built-in function
- The %LOOKUPLT built-in function
- The %LOOKUPLE built-in function
- The %LOOKUPGT built-in function
- The %LOOKUPGE built-in function

For more information about the LOOKUP operation code, see:
- "Searching an Array with an Index" on page 181
- "Searching an Array without an Index"
- "LOOKUP (Look Up a Table or Array Element)" on page 599

For more information about the %LOOKUPxx built-in functions, see "%LOOKUPxx (Look Up an Array Element)" on page 455.

Searching an Array without an Index

When searching an array without an index, use the status (on or off) of the resulting indicators to determine whether a particular element is present in the array. Searching an array without an index can be used for validity checking of input data to determine if a field is in a list of array elements. Generally, an equal LOOKUP is used.

In factor 1 in the calculation specifications, specify the search argument (data for which you want to find a match in the array named) and place the array name factor 2.

In factor 2 specify the name of the array to be searched. At least one resulting indicator must be specified. Entries must not be made in both high and low for the same LOOKUP operation. The resulting indicators must not be specified in high or

low if the array is not in sequence (ASCEND or DESCEND keywords). Control level and conditioning indicators (specified in positions 7 through 11) can also be used. The result field cannot be used.

The search starts at the beginning of the array and ends at the end of the array or when the conditions of the lookup are satisfied. Whenever an array element is found that satisfies the type of search being made (equal, high, low), the resulting indicator is set on.

Figure 68 shows an example of a LOOKUP on an array without an index.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++++
FARRFILE   IT   F    5        DISK
F*
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
DDPTNOS           S              5S 0 DIM(50) FROMFILE(ARRFILE)
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* The LOOKUP operation is processed and, if an element of DPTNOS equal
C* to the search argument (DPTNUM) is found, indicator 20 is set on.
C     DPTNUM        LOOKUP    DPTNOS                                  20
```

Figure 68. LOOKUP Operation for an Array without an Index

ARRFILE, which contains department numbers, is defined in the file description specifications as an input file (I in position 17) with an array file designation (T in position 18). The file is program described (F in position 22), and each record is 5 positions in length (5 in position 27).

In the definition specifications, ARRFILE is defined as containing the array DPTNOS. The array contains 50 entries (DIM(50)). Each entry is 5 positions in length (positions 33-39) with zero decimal positions (positions 41-42). One department number can be contained in each record (PERRCD defaults to 1).

Searching an Array with an Index

To find out which element satisfies a LOOKUP search, start the search at a particular element in the array. To do this type of search, make the entries in the calculation specifications as you would for an array without an index. However, in factor 2, enter the name of the array to be searched, followed by a parenthesized numeric field (with zero decimal positions) containing the number of the element at which the search is to start. This numeric constant or field is called the index because it points to a certain element in the array. The index is updated with the element number which satisfied the search or is set to 0 if the search failed.

You can use a numeric constant as the index to test for the existence of an element that satisfies the search starting at an element other than 1.

All other rules that apply to an array without an index apply to an array with an index.

Figure 69 on page 182 shows a LOOKUP on an array with an index. This example shows the same array of department numbers, DPTNOS, as Figure 68. However, an alternating array of department descriptions, DPTDSC, is also defined. Each element in DPTDSC is 20 positions in length. If there is insufficient data in the file

to initialize the entire array, the remaining elements in DPTNOS are filled with zeros and the remaining elements in DPTDSC are filled with blanks.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++++
FARRFILE   IT   F   25            DISK
F*
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
DDPTNOS           S              5S 0 DIM(50) FROMFILE(ARRFILE)
DDPTDSC           S             20A   DIM(50) ALT(DPTNOS)
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* The Z-ADD operation begins the LOOKUP at the first element in DPTNOS.
C                   Z-ADD     1             X                 3 0
C* At the end of a successful LOOKUP, when an element has been found
C* that contains an entry equal to the search argument DPTNUM,
C* indicator 20 is set on and the MOVE operation places the department
C* description, corresponding to the department number, into DPTNAM.
C     DPTNUM        LOOKUP    DPTNOS(X)                              20
C* If an element is found that is equal to the search argument,
C* element X of DPTDSC is moved to DPTNAM.
C                   IF            *IN20
C                   MOVE      DPTDSC(X)     DPTNAM           20
C                   ENDIF
```

Figure 69. LOOKUP Operation on an Array with an Index

Using Arrays

Arrays can be used in input, output, or calculation specifications.

Specifying an Array in Calculations

An entire array or individual elements in an array can be specified in calculation specifications. Individual elements are processed like fields.

A noncontiguous array defined with the OVERLAY keyword cannot be used with the MOVEA operation or in the result field of a PARM operation.

To specify an entire array, use only the array name, which can be used as factor 1, factor 2, or the result field. The following operations can be used with an array name:

ADD	ADDDUR	CHECK	CHECKR	CLEAR
DEFINE	DIV	EVAL	EXTRCT	LOOKUP
MOVE	MOVEL	MOVEA	MULT	PARM
RESET	SCAN	SORTA	SQRT	SUB
SUBDUR	XFOOT	Z-ADD	Z-SUB	

Several other operations can be used with an array element only but not with the array name alone. These operations include but are not limited to:

BITON	BITOFF	CABxx	CAT	COMP
DO	DOU	DOUxx	DOW	DOWxx
IF	IFxx	MVR	SUBST	TESTB
TESTN	TESTZ	WHEN	WHENxx	

When specified with an array name without an index or with an asterisk as the index (for example, ARRAY or ARRAY(*)) certain operations are repeated for each element in the array. These are:

ADD	ADDDUR	DIV	EVAL	EXTRCT
MOVE	MOVEL	MULT	SQRT	SUB
Z-ADD	Z-SUB			

The following rules apply to these operations when an array name without an index is specified:

- When factor 1, factor 2, and the result field are arrays with the same number of elements, the operation uses the first element from every array, then the second element from every array until all elements in the arrays are processed. If the arrays do not have the same number of entries, the operation ends when the last element of the array with the fewest elements has been processed. When factor 1 is not specified for the ADD, SUB, MULT, and DIV operations, factor 1 is assumed to be the same as the result field.
- When one of the factors is a field, a literal, or a figurative constant and the other factor and the result field are arrays, the operation is done once for every element in the shorter array. The same field, literal, or figurative constant is used in all of the operations.
- The result field must always be an array.

- If an operation code uses factor 2 only (for example, Z-ADD, Z-SUB, SQRT, ADD, SUB, MULT, or DIV may not have factor 1 specified) and the result field is an array, the operation is done once for every element in the array. The same field or constant is used in all of the operations if factor 2 is not an array.
- Resulting indicators (positions 71 through 76) cannot be used because of the number of operations being processed.

Note: When used in an EVAL operation, %ADDR(arr) and %ADDR(arr(*)) do not have the same meaning. See "%ADDR (Get Address of Variable)" on page 406 for more details.

When coding an EVAL or a SORTA operation, built-in function %SUBARR(arr) can be used to select a portion of the array to be used in the operation. See "%SUBARR (Set/Get Portion of an Array)" on page 480 for more detail.

Sorting Arrays

You can sort arrays or a section of an array using the SORTA operation code. The array is sorted into sequence (ascending or descending), depending on the sequence specified for the array on the definition specification.

Sorting using Part of the Array as a Key

You can use the OVERLAY keyword to overlay one array over another. For example, you can have a base array which contains names and salaries and two overlay arrays (one for the names and one for the salaries). You could then sort the base array by either name or salary by sorting on the appropriate overlay array.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D                 DS
D Emp_Info                      50    DIM(500) ASCEND
D   Emp_Name                    45    OVERLAY(Emp_Info:1)
D   Emp_Salary                   9P 2 OVERLAY(Emp_Info:46)
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C* The following SORTA sorts Emp_Info by employee name.
C* The sequence of Emp_Name is used to determine the order of the
C* elements of Emp_Info.
C                   SORTA     Emp_Name
C* The following SORTA sorts Emp_Info by employee salary
C* The sequence of Emp_Salary is used to determine the order of the
C* elements of Emp_Info.
C                   SORTA     Emp_Salary
```

Figure 70. SORTA Operation with OVERLAY

Array Output

Entire arrays can be written out only at the end of the program when the LR indicator has been set on. To indicate that an entire array is to be written out, specify the name of the output file with the TOFILE keyword on the definition specifications. This file must be described as a sequentially organized combined file in the file description specifications.

If the file is a combined file and is externally described as a physical file, the information in the array at the end of the program replaces the information read into the array at the start of the program. Logical files may give unpredictable results.

If an entire array is to be written to an output record (using output specifications), describe the array along with any other fields for the record:

- Positions 30 through 43 of the output specifications must contain the array name used in the definition specifications.
- Positions 47 through 51 of the output specifications must contain the record position where the last element of the array is to end. If an edit code is specified, the end position must include blank positions and any extensions due to the edit code (see "Editing Entire Arrays" listed next in this section).

Output indicators (positions 21 through 29) can be specified. Zero suppress (position 44), blank-after (position 45), and data format (position 52) entries pertain to every element in the array.

Editing Entire Arrays

When editing is specified for an entire array, all elements of the array are edited. If different editing is required for various elements, refer to them individually.

When an edit code is specified for an entire array (position 44), two blanks are automatically inserted between elements in the array: there are blanks to the left of every element in the array except the first. When an edit word is specified, the blanks are not inserted. The edit word must contain all the blanks to be inserted.

Using Dynamically-Sized Arrays

If you don't know the number of elements you will need in an array until runtime, you can define the array with the maximum size, and then use a subset of the array in your program.

To do this, you use the %SUBARR builtin function to control which elements are used when you want to work with all the elements of your array in one operation. You can also use the %LOOKUP builtin function to search part of your array.

```
 * Define the "names" array as large as you think it could grow
D names           S             25A   VARYING DIM(2000)
 * Define a variable to keep track of the number of valid elements
D numNames        S             10I 0 INZ(0)
 * Define another array
D temp            S             50A   DIM(20)
D p               S             10I 0
 /free
     // set 3 elements in the names array
     names(1) = 'Friendly';
     names(2) = 'Rusty';
     names(3) = 'Jerome';
     names(4) = 'Tom';
     names(5) = 'Jane';
     numNames = 5;

     // copy the current names to the temporary array
     // Note: %subarr could also be used for temp, but
     //       it would not affect the number of elements
     //       copied to temp
     temp = %subarr(names : 1 : numNames);

     // change one of the temporary values, and then copy
     // the changed part of the array back to the "names" array
     temp(3) = 'Jerry';
     temp(4) = 'Harry';
     // The number of elements actually assigned will be the
     // minimum of the number of elements in any array or
     // subarray in the expression.  In this case, the
     // available sizes are 2 for the "names" sub-array,
     // and 18 for the "temp" subarray, from element 3
     // to the end of the array.
     %subarr(names : 3 : 2) = %subarr(temp : 3);
     // sort the "names" array
     sorta %subarr(names : 1 : numNames);

     // search the "names" array
     // Note: %SUBARR is not used with %LOOKUP.  Instead,
     //       the start element and number of elements
     //       are specified in the third and fourth
     //       parameters of %LOOKUP.
     p = %lookup('Jane' : names : 1 : numNames);
```

Figure 71. Example using a dynamically-sized array

Tables

The explanation of arrays applies to tables except for the following differences:

Activity	Differences
Defining	A table name must be a unique symbolic name that begins with the letters TAB.
Using and Modifying Table Elements	Only one element of a table is active at one time. The table name is used to refer to the active element.
Searching	The LOOKUP operation is specified differently for tables. Different built-in functions are used for searching tables.

Note: You cannot define a table in a subprocedure.

The following can be used to search a table:

- The LOOKUP operation code
- The %TLOOKUP built-in function
- The %TLOOKUPLT built-in function
- The %TLOOKUPLE built-in function
- The %TLOOKUPGT built-in function
- The %TLOOKUPGE built-in function

For more information about the LOOKUP operation code, see:
- "LOOKUP with One Table"
- "LOOKUP with Two Tables" on page 188
- "LOOKUP (Look Up a Table or Array Element)" on page 599

For more information about the %TLOOKUPxx built-in functions, see "%TLOOKUPxx (Look Up a Table Element)" on page 489.

LOOKUP with One Table

When a single table is searched, factor 1, factor 2, and at least one resulting indicator must be specified. Conditioning indicators (specified in positions 7 through 11) can also be used.

Whenever a table element is found that satisfies the type of search being made (equal, high, low), the table element is made the current element for the table. If the search is not successful, the previous current element remains the current element.

Before a first successful LOOKUP, the first element is the current element.

Resulting indicators reflect the result of the search. If the indicator is on, reflecting a successful search, the element satisfying the search is the current element.

LOOKUP with Two Tables

When two tables are used in a search, only one is actually searched. When the search condition (high, low, equal) is satisfied, the corresponding elements are made available for use.

Factor 1 must contain the search argument, and factor 2 must contain the name of the table to be searched. The result field must name the table from which data is also made available for use. A resulting indicator must also be used. Control level and conditioning indicators can be specified in positions 7 through 11, if needed.

The two tables used should have the same number of entries. If the table that is searched contains more elements than the second table, it is possible to satisfy the search condition. However, there might not be an element in the second table that corresponds to the element found in the search table. Undesirable results can occur.

Note: If you specify a table name in an operation other than LOOKUP before a successful LOOKUP occurs, the table is set to its first element.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* The LOOKUP operation searches TABEMP for an entry that is equal to
C* the contents of the field named EMPNUM. If an equal entry is
C* found in TABEMP, indicator 09 is set on, and the TABEMP entry and
C* its related entry in TABPAY are made the current elements.
C     EMPNUM        LOOKUP    TABEMP        TABPAY                  09
C* If indicator 09 is set on, the contents of the field named
C* HRSWKD are multiplied by the value of the current element of
C* TABPAY.
C                   IF        *IN09
C     HRSWKD        MULT(H)   TABPAY        AMT              6 2
C                   ENDIF
```

Figure 72. Searching for an Equal Entry

Specifying the Table Element Found in a LOOKUP Operation

Whenever a table name is used in an operation other than LOOKUP, the table name actually refers to the data retrieved by the last successful search. Therefore, when the table name is specified in this fashion, elements from a table can be used in calculation operations.

If the table is used as factor 1 in a LOOKUP operation, the current element is used as the search argument. In this way an element from a table can itself become a search argument.

The table can also be used as the result field in operations other than the LOOKUP operation. In this case the value of the current element is changed by the calculation specification. In this way the contents of the table can be modified by calculation operations. See Figure 73.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C     ARGMNT        LOOKUP    TABLEA                                 20
C* If element is found multiply by 1.5
C* If the contents of the entire table before the MULT operation
C* were 1323.5, -7.8, and 113.4  and the value of ARGMNT is -7.8,
C* then the second element is the current element.
C* After the MULT operation, the entire table now has the
C* following value:  1323.5, -11.7, and 113.4.
C* Note that only the second element has changed since that was
C* the current element, set by the LOOKUP.
C                   IF        *IN20
C     TABLEA        MULT      1.5           TABLEA
C                   ENDIF
```

Figure 73. Specifying the Table Element Found in LOOKUP Operations

Chapter 13. Editing Numeric Fields

Editing provides a means of punctuating numeric fields, including the printing of currency symbols, commas, periods, minus signs, and floating minus. It also provides for field sign movement from the rightmost digit to the end of the field, blanking zero fields, spacing in arrays, date field editing, and currency symbol or asterisk protection.

A field can be edited by edit codes or edit words. You can print fields in an edited format using output specifications, or you can obtain the edited value of the field in calculation specifications using the built-in functions %EDITC (edit code) and %EDITW (edit word).

Note: For a description of how to edit Entry field parts and Static text parts, see *Programming with VisualAge RPG*, SC09-2449-05.

When you print fields that are not edited, the fields appear exactly as they are internally represented. The following examples show why you may want to edit numeric output fields.

Type of Field	Field in the Computer	Printing of Unedited Field	Printing of Edited Field
Alphanumeric	JOHN T SMITH	JOHN T SMITH	JOHN T SMITH
Numeric (positive)	0047652	0047652	47652
Numeric (negative)	004765r	004765r	47652-

The unedited alphanumeric field and the unedited positive numeric field are easy to read when printed, but the unedited negative numeric field is confusing because it contains a 'r', which is not numeric. The 'r' is a combination of the digit 2 and the negative sign for the field. They are combined so that one of the positions of the field does not have to be set aside for the sign. The combination is convenient for storing the field in the computer, but it makes the output hard to read. Numeric fields must be edited before they are printed.

Edit Codes

Edit codes provide a means of editing numeric fields according to a predefined pattern. They are divided into two categories: simple (X, Y, Z) and combination (1 through 4, A through D, J through Q). You enter the edit code in position 44 of the output specifications for the field to be edited. Or, you can specify the edit code as the second parameter of the %EDITC built-in function in calculation specifications.

Simple Edit Codes

You can use simple edit codes to edit numeric fields without having to specify any punctuation. These codes and their functions are:

- The X edit code ensures a hexadecimal 3 sign for positive fields. However, because the system does this, you normally do not have to specify this code. Leading zeros are not suppressed. The X edit code does not modify negative numbers.
- The Y edit code is normally used to edit a 3 to 9 digit date field. It suppresses the leftmost zeros of date fields, up to but not including the digit preceding the first separator. Slashes are inserted to separate the day, month, and year. The DATEDIT(fmt{separator}) and DECEDIT('value') keywords on the control specification can be used to alter edit formats.
- The Y edit code is not valid for *YEAR, *MONTH, and *DAY.
- The Z edit code removes the sign (plus or minus) from and suppresses the leading zeros of a numeric field. The decimal point is not placed in the field and is not printed.

Combination Edit Codes

The combination edit codes (1 through 4, A through D, J through Q) punctuate a numeric field.

The DECEDIT keyword on the control specification determines what character is used for the decimal separator and whether leading zeroes are suppressed. The decimal position of the source field determines whether and where a decimal point is printed. If decimal positions are specified for the source field and the zero balance is to be suppressed, the decimal separator prints only if the field is not zero. If a zero balance is not to be printed, a zero field prints as blanks.

When a zero balance is to be printed and the field is equal to zero, either of the following is printed:
- A decimal separator followed by n zeros, where n is the number of decimal places in the field
- A zero in the units position of a field if no decimal places are specified.

You can use a floating currency symbol or asterisk protection with any of the 12 combination edit codes. To specify a floating currency symbol, code the currency symbol in positions 53-55 on the output specification, along with an edit code in position 44 for the field to be edited. The floating currency symbol appears to the left of the first significant digit. The floating currency symbol does not print on a zero balance when an edit code is used that suppresses the zero balance. The currency symbol does not appear on a zero balance when an edit code is used that suppresses the zero balance.

A dollar sign ($) is used as the currency symbol unless a currency symbol is specified with the CURSYM keyword on the control specification.)

An asterisk constant coded in positions 53 through 55 of the output specifications ('*'), along with an edit code for the field to be edited causes an asterisk to be printed for each zero suppressed. A complete field of asterisks is printed on a zero balance source field. To specify asterisk protection using the built-in function %EDITC, specify *ASTFILL as the third parameter.

Asterisk fill and the floating currency symbol cannot be used with the simple (X, Y, Z) edit codes.

For the built-in function %EDITC, you specify a floating currency symbol in the third parameter. To use the currency symbol for the program, specify *CURSYM. To use another currency symbol, specify a character constant of length 1.

A currency symbol can appear before the asterisk fill (fixed currency symbol). This requires two output specifications with the following coding:

1. Place a currency symbol constant in position 53 of the first output specification. The end position specified in positions 47-51 should be one space before the beginning of the edited field.
2. In the second output specification, place the edit field in positions 30-43, an edit code in position 44, end position of the edit field in positions 47-51, and '*' in positions 53-55.

   You can do this using the %EDITC built-in function by concatenating the currency symbol to the %EDITC result as follows

```
C           EVAL       X = '$' + %EDITC(N: 'A' : *ASTFILL)
```

When an edit code is used to print an entire array, two blanks precede each element of the array (except the first element).

Note: You cannot edit an array using the %EDITC built-in function.

Table 20 summarizes the functions of the combination edit codes. The codes edit the field in the format listed on the left. A negative field can be punctuated with no sign, CR, a minus sign (–), or a floating minus sign as shown on the top of the figure.

Table 20. Combination Edit Codes

		Negative Balance Indicator			
Prints with Grouping Separator	**Prints Zero Balance**	**No Sign**	**CR**	**–**	**Floating Minus**
Yes	Yes	1	A	J	N
Yes	No	2	B	K	0
No	Yes	3	C	L	P
No	No	4	D	M	Q

Editing Considerations

When you specify any of the edit codes, do the following:

- Edit fields of a non-printer file with caution. If you do edit fields of a non-printer file, be aware of the contents of the edited fields and the effects of any operations you do on them. For example, if you use the file as input, the fields written out with editing must be considered character fields, not numeric fields.
- Consideration should be given to data added by the edit operation. The amount of punctuation added increases the overall length of the output field. If these added characters are not considered when editing in output specifications, the output fields may overlap.
- The end position specified for output is the end position of the edited field. For example, if any of the edit codes J through M are specified, the end position is the position of the minus sign (or blank if the field is positive).
- The compiler assigns a character position for the sign even for unsigned numeric fields.

Summary of Edit Codes

Table 21 summarizes the edit codes and the options they provide. A simplified version of this table is printed above positions 45 through 70 on the output specifications. Table 22 on page 195 shows how fields look after they are edited.

Table 23 on page 196 shows the effect that the different edit codes have on the same field with a specified end position for output.

Table 21. Edit Codes

				DECEDIT Keyword Parameter				
Edit Code	**Commas**	**Decimal Point**	**Sign for Negative Balance**	**'.'**	**','**	**'0,'**	**'0.'**	**Zero Suppress**
1	Yes	Yes	No Sign	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
2	Yes	Yes	No Sign	Blanks	Blanks	Blanks	Blanks	Yes
3		Yes	No Sign	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
4		Yes	No Sign	Blanks	Blanks	Blanks	Blanks	Yes
A	Yes	Yes	CR	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
B	Yes	Yes	CR	Blanks	Blanks	Blanks	Blanks	Yes
C		Yes	CR	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
D		Yes	CR	Blanks	Blanks	Blanks	Blanks	Yes
J	Yes	Yes	– 5 (minus)	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
K	Yes	Yes	– (minus)	Blanks	Blanks	Blanks	Blanks	Yes
L		Yes	– (minus)	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
M		Yes	– (minus)	Blanks	Blanks	Blanks	Blanks	Yes
N	Yes	Yes	– (floating minus)	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
O	Yes	Yes	– (floating minus)	Blanks	Blanks	Blanks	Blanks	Yes
P		Yes	– (floating minus)	.00 or 0	,00 or 0	0,00 or 0	0.00 or 0	Yes
Q		Yes	– (floating minus)	Blanks	Blanks	Blanks	Blanks	Yes
X[1]								Yes
Y[2]								Yes
Z[3]								Yes

Table 21. Edit Codes (continued)

				DECEDIT Keyword Parameter				
Edit Code	**Commas**	**Decimal Point**	**Sign for Negative Balance**	**'.'**	**','**	**'0,'**	**'0.'**	**Zero Suppress**

[1]The X edit code ensures a hexadecimal 3 sign for positive values. Because the system does this for you, normally you do not have to specify this code.

[2]The Y edit code suppresses the leftmost zeros of date fields, up to but not including the digit preceding the first separator. The Y edit code also inserts slashes (/) between the month, day, and year according to the following pattern:

```
   nn/n
   nn/nn
   nn/nn/n
   nn/nn/nn
  nnn/nn/nn
   nn/nn/nnnn
  nnn/nn/nnnn
 nnnn/nn/nn
nnnnn/nn/nn
```

[3]The Z edit code removes the sign (plus or minus) from a numeric field and suppresses leading zeros.

Table 22. Examples of Edit Code Usage

Edit Codes	**Positive Number-Two Decimal Positions**	**Positive Number-No Decimal Positions**	**Negative Number-Three Decimal Positions**	**Negative Number-No Decimal Positions**	**Zero Balance-Two Decimal Positions**	**Zero Balance-No Decimal Positions**
Unedited	1234567	1234567	00012b[4]	000000	000000	
1	12,345.67	1,234,567	.120	120	.00	0
2	12,345.67	1,234,567	.120	120		
3	12345.67	1234567	.120	120	.00	0
4	12345.67	1234567	.120	120		
A	12,345.67	1,234,567	.120CR	120CR	.00	0
B	12.345.67	1,234,567	.120CR	120CR		
C	12345.67	1234567	.120CR	120CR	.00	0
D	12345.67	1234567	.120CR	120CR		
J	12,345.67	1,234,567	.120–	120–	.00	0
K	12,345,67	1,234,567	.120–	120–		
L	12345.67	1234567	.120–	120–	.00	0
M	12345.67	1234567	.120–	120–		
N	12,345.67	1,234,567	–.120	–120	.00	0
O	12,345,67	1,234,567	–.120	–120		
P	12345.67	1234567	–.120	–120	.00	0
Q	12345.67	1234567	–.120	–120		
X[1]	1234567	1234567	00012b[4]	000000	000000	
Y[2]			0/01/20	0/01/20	0/00/00	0/00/00
Z[3]	1234567	1234567	120	120		

Table 22. Examples of Edit Code Usage (continued)

Edit Codes	Positive Number-Two Decimal Positions	Positive Number-No Decimal Positions	Negative Number-Three Decimal Positions	Negative Number-No Decimal Positions	Zero Balance-Two Decimal Positions	Zero Balance-No Decimal Positions

[1] The X edit code ensures a hex F sign for positive values. Because the system does this for you, normally you do not have to specify this code.

[2] The Y edit code suppresses the leftmost zeros of date fields, up to but not including the digit preceding the first separator. The Y edit code also inserts slashes (/) between the month, day, and year according to the following pattern:

```
   nn/n
   nn/nn
   nn/nn/n
   nn/nn/nn
  nnn/nn/nn
   nn/nn/nnnn  Format used with M, D or blank in position 19
  nnn/nn/nnnn  Format used with M, D or blank in position 19
 nnnn/nn/nn    Format used with Y in position 19
nnnnn/nn/nn    Format used with Y in position 19
```

[3] The Z edit code removes the sign (plus or minus) from a numeric field and suppresses leading zeros of a numeric field.

[4] The ␣ represents a blank. This may occur if a negative zero does not correspond to a printable character.

Table 23. Effects of Edit Codes on End Position

	Negative Number, 2 Decimal Positions. End Position Specified as 10.								
	Output Print Positions								
Edit Code	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
[1]r represents a negative 2.									
Unedited				0	0	4	1	r[1]	
1					4	.	1	2	
2					4	.	1	2	
3					4	.	1	2	
4					4	.	1	2	
A			4	.	1	2	C	R	
B			4	.	1	2	C	R	
C			4	.	1	2	C	R	
D			4	.	1	2	C	R	
J				4	.	1	2	–	
r				4	.	1	2	–	
L				4	.	1	2	–	
M				4	.	1	2	–	
N				–	4	.	1	2	
O				–	4	.	1	2	
P				–	4	.	1	2	

Table 23. Effects of Edit Codes on End Position (continued)

	Negative Number, 2 Decimal Positions. End Position Specified as 10.								
	Output Print Positions								
Edit Code	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
Q				–	4	.	1	2	
X				0	0	4	1	r[1]	
Y			0	/	4	1	/	2	
Z						4	1	2	

Edit Words

If you have editing requirements that cannot be met by using the edit codes, you can use an edit word. An edit word is a character literal or a named constant specified in positions 53 - 80 of the output specification. It describes the editing pattern for a numeric and allows you to directly specify:

- Blank spaces
- Commas and decimal points, and their position
- Suppression of unwanted zeros
- Leading asterisks
- The currency symbol and its position
- Addition of constant characters
- Output of the negative sign, or CR, as a negative indicator.

The edit word is used as a template that the system applies to the source data to produce the output.

The edit word can be specified directly on an output specification or can be specified as a named constant with a named constant name appearing in the edit word field of the output specification. You can obtain the edited value of the field in calculation specifications using the built-in function %EDITW (edit word).

Edit words are limited to 115 characters.

How to Code an Edit Word

To use an edit word, code the output specifications as shown below:

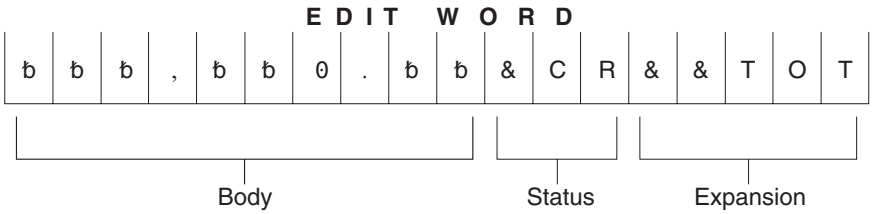
Position	**Entry**
21-29	Can contain conditioning indicators.
30-43	Contains the name of the numeric field from which the data that is to be edited is taken.
44	Edit code: Must be blank, if you are using an edit word to edit the source data.
45	A “B” in this position indicates that the source data is to be set to zero or blanks after it has been edited and output. Otherwise, the source data remains unchanged.
47-51	Identifies the end (rightmost) position of the field in the output record.
53-80	Edit word: Can be up to 26 characters long and must be enclosed by apostrophes, unless it is a named constant. Enter the leading

apostrophe, or begin the named constant name in column 53. The edit word, unless a named constant, must begin in column 54.

To edit using an edit word in calculation specifications, use built-in function %EDITW, specifying the value to be edited as the first parameter, and the edit word as the second parameter

Parts of an Edit Word

An edit word (coded into positions 53 to 80 of the output specifications) consists of three parts: the body, the status, and the expansion. The following shows the three parts of an edit word:



The body is the space for the digits transferred from the source data field to the output record. The body begins at the leftmost position of the edit word. The number of blanks (plus one zero or an asterisk) in the edit word body must be equal to or greater than the number of digits of the source data field to be edited. The body ends with the rightmost character that can be replaced by a digit.

The status defines a space to allow for a negative indicator, either the two letters CR or a minus sign (–). The negative indicator specified is output only if the source data is negative. All characters in the edit word between the last replaceable character (blank, zero suppression character) and the negative indicator are also output with the negative indicator only if the source data is negative; if the source data is positive, these status positions are replaced by blanks. Edit words without the CR or – indicators have no status positions.

The status must be entered after the last blank in the edit word. If more than one CR follows the last blank, only the first CR is treated as a status; the remaining CRs are treated as constants. For the minus sign to be considered as a status, it must be the last character in the edit word.

The expansion is a series of ampersands and constant characters entered after the status. Ampersands are replaced by blank spaces in the output; constants are output as is. If status is not specified, the expansion follows the body.

Forming the Body of an Edit Word

The following characters have special meanings when used in the body of an edit word.

Ampersand: Causes a blank in the edited field. The example below might be used to edit a telephone number. Note that the zero in the first position is required to print the constant AREA.

Edit Word	Source Data	Appears in Output Record as:
'0AREA&␣␣␣&NO.&␣␣␣-␣␣␣␣'	4165551212	␣AREA␣416␣NO.␣555-1212

Asterisk: The first asterisk in the body of an edit word also ends zero suppression. Subsequent asterisks put into the edit word are treated as constants (see Constants below). Any zeros in the edit word following this asterisk are also treated as constants. There can be only one end-zero-suppression character in an edit word, and that character is the first asterisk *or* the first zero in the edit word.

If an asterisk is used as an end-zero-suppression character, all leading zeros that are suppressed are replaced with asterisks in the output. Otherwise, the asterisk suppresses leading zeros in the same way as described below for Zeros.

Edit Word	Source Data	Appears in Output Record as:
'*␣␣␣␣␣␣.␣␣'	000000123	*000001.23
'␣␣␣␣␣*␣.␣␣'	000000000	******0.00
'␣␣␣␣␣*␣.␣␣**'	000056342	****563.42**

Note that leading zeros appearing after the asterisk position are output as leading zeros. Only the suppressed leading zeros, including the one in the asterisk position, are replaced by asterisks.

Blank: Blank is replaced with the character from the corresponding position of the source data field specified by the field name in positions 30 through 43 of the output specifications. A blank position is referred to as a digit position.

Constants: All other characters entered into the body of the edit word are treated as constants. If the source data is such that the output places significant digits or leading zeros to the left of any constant, then that constant appears in the output. Otherwise, the constant is suppressed in the output. Commas and the decimal point follow the same rules as for constants. Notice in the examples below, that the presence of the end-zero-suppression character as well as the number of significant digits in the source data, influence the output of constants.

The following edit words could be used to print cheques. Note that the second asterisk is treated as a constant, and that, in the third example, the constants preceding the first significant digit are not output.

Edit Word	Source Data	Appears in Output Record as:
'$␣␣␣␣␣**DOLLARS&␣␣&CTS'	000012345	$****123*DOLLARS␣45␣CTS
'$␣␣␣␣␣**DOLLARS&␣␣&CTS'	000000006	$********DOLLARS␣06␣CTS
'$␣␣␣␣␣␣␣&DOLLARS&␣␣&CTS'	000000006	$␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣6␣CTS

A date could be printed by using either edit word:

Edit Word	Source Data	Appears in Output Record as:
'bb/bb/bb'	010388	b1/03/88
'0bb/bb/bb'	010389	b01/03/89

Note that any zeros or asterisks following the first occurrence of an edit word are treated as constants. The same is true for – and CR:

Edit Word	Source Data	Appears in Output Record as:
'bb0.bb000'	01234	b12.34000
'bb*.bb000'	01234	*12.34000

Currency Symbol: A currency symbol followed directly by a first zero in the edit word (end-zero-suppression character) is said to float. All leading zeros are suppressed in the output and the currency symbol appears in the output immediately to the left of the most significant digit.

Edit Word	Source Data	Appears in Output Record as:
'bb,bbb,b$0.bb'	000000012	bbbbbbbbb$.12
'bb,bbb,b$0.bb'	000123456	bbbb$1,234.56

If the currency symbol is put into the first position of the edit word, then it will always appear in that position in the output. This is called a fixed currency symbol.

Edit Word	Source Data	Appears in Output Record as:
'$b,bbb,bb0.bb'	000123456	$bbbb1,234.56
'$bb,bbb,0b0.bb'	000000000	$bbbbbbbb00.00
'$b,bbb,*bb.bb'	000123456	$****1,234.56

A currency symbol anywhere else in the edit word and not immediately followed by a zero end-suppression-character is treated as a constant (see Constants above).

Decimals and Commas: Decimals and commas are in the same relative position in the edited output field as they are in the edit word unless they appear to the left of the first significant digit in the edit word. In that case, they are blanked out or replaced by an asterisk.

In the following examples below, all the leading zeros will be suppressed (default) and the decimal point will not print unless there is a significant digit to its left.

Edit Word	Source Data	Appears in Output Record as:
'bbbbbbb'	0000072	bbbbb72
'bbbbbbb.bb'	000000012	bbbbbbbb12
'bbbbbbb.bb'	000000123	bbbbbb1.23

Zeros: The first zero in the body of the edit word is interpreted as an end-zero-suppression character. This zero is placed where zero suppression is to end. Subsequent zeros put into the edit word are treated as constants (see Constants above).

Any leading zeros in the source data are suppressed up to and including the position of the end-zero-suppression character. Significant digits that would appear in the end-zero-suppression character position, or to the left of it, are output.

Edit Word	Source Data	Appears in Output Record as:
'bbb0bbbbbb'	00000004	bbbb000004
'bbb0bbbbbb'	012345	bbbb012345
'bbb0bbbbbb'	012345678	bb12345678

If the leading zeros include, or extend to the right of, the end-zero-suppression character position, that position is replaced with a blank. This means that if you wish the same number of leading zeros to appear in the output as exist in the source data, the edit word body must be wider than the source data.

Edit Word	Source Data	Appears in Output Record as:
'0bbb'	0156	b156
'0bbbb'	0156	b0156

Constants (including commas and decimal point) that are placed to the right of the end-zero-suppression character are output, even if there is no source data.

Constants to the left of the end-zero-suppression character are only output if the source data has significant digits that would be placed to the left of these constants.

Edit Word	Source Data	Appears in Output Record as:
'bbbbbb0.bb'	000000001	bbbbbb.01
'bbbbbb0.bb'	000000000	bbbbbb.00
'bbb,b0b.bb'	00000012	bbbbb0.12
'bbb,b0b.bb'	00000123	bbbbb1.23
'b0b,bbb.bb'	00000123	bb0,001.23

Forming the Status of an Edit Word

The following characters have special meanings when used in the status of an edit word:

Ampersand: Causes a blank in the edited output field. An ampersand cannot be placed in the edited output field.

CR or minus symbol: If the sign in the edited output is plus (+), these positions are blanked out. If the sign in the edited output field is minus (–), these positions remain undisturbed.

The following example adds a negative value indication. The minus sign will print only when the value in the field is negative. A CR symbol fills the same function as a minus sign.

Edit Word	Source Data	Appears in Output Record as:
'bbbbbbb.bb–'	000000123–	bbbbb1.23–
'bbbbbbb.bb–'	000000123	bbbbb1.23b

Constants between the last replaceable character and the – or CR symbol will print only if the field is negative; otherwise, blanks will print in these positions. Note the use of ampersands to represent blanks:

Edit Word	Source Data	Appears in Output Record as:
'b,bbb,bb0.bb&30&DAY&CR'	000000123–	bbbbbbbbb1.23b30bDAYbCR
'b,bbb,bb0.bb&30&DAY&CR'	000000123	bbbbbbbbb1.23bbbbbbbbbb

Formatting the Expansion of an Edit Word

The characters in the expansion portion of an edit word are always written. The expansion cannot contain blanks. If a blank is required in the edited output field, specify an ampersand in the body of the edit word.

Constants may be added to print on every line:

Edit Word	Source Data	Appears in Output Record as:
'b,bb0.bb&CR&NET'	000123–	bbbb1.23bCRbNET
'b,bb0.bb&CR&NET'	000123	bbbb1.23bbbbNET

Note that the CR in the middle of a word may be detected as a negative field value indication. If a word such as SECRET is required, use the coding in the example below.

Edit Word	Source Data	Appears in Output Record as:
'␣␣0.␣␣&SECRET'	12345–	123.45␣SECRET
'␣␣0.␣␣&SECRET'	12345	123.45␣␣␣␣␣ET
'␣␣0.␣␣&CR&&SECRET'	12345	123.45␣␣␣␣␣SECRET

Summary of Coding Rules for Edit Words

The following rules apply to edit words:

- Position 44 (edit codes) must be blank.
- Positions 30 through 43 (field name) must contain the name of a numeric field.
- An edit word must be enclosed in apostrophes, unless it is a named constant. Enter the leading apostrophe or begin a named constant name in position 53. The edit word itself must begin in position 54.
- The edit word can contain more digit positions (blanks plus the initial zero or asterisk) than the field to be edited, but must not contain less. If there are more digit positions in the edit word than there are digits in the field to be edited, leading zeros are added to the field before editing.
- If leading zeros from the source data are desired, the edit word must contain one more position than the field to be edited, and a zero must be placed in the high-order position of the edit word.
- In the body of the edit word only blanks and the zero-suppression stop characters (zero and asterisk) are counted as digit positions. The floating currency symbol is not counted as a digit position.
- When the floating currency symbol is used, the sum of the number of blanks and the zero-suppression stop character (digit positions) contained in the edit word must be equal to or greater than the number of positions in the field to be edited.
- Any zeros or asterisks following the leftmost zero or asterisk are treated as constants; they are not replaceable characters.
- When editing an unsigned integer field, DB and CR are allowed and will always print as blanks.

Editing Externally Described Files

To edit output for remote disk files, edit codes must be specified in data description specifications (DDS).

Note: Edit codes cannot be used for special files.

Chapter 14. Initialization of Data

This section describes how data is initialized. Initialization of data consists of three parts: the initialization subroutine (*INZSR), the CLEAR and RESET operation codes, and data initialization (INZ keyword). For information on initializing components, see "Initializing Components" on page 31.

Initialization Subroutine (*INZSR)

The initialization subroutine allows you to process calculation specifications. It is declared like any other subroutine, but with *INZSR in factor 1. You can enter any operations in this subroutine except the RESET operation. *INZSR can also be called explicitly by using an EXSR or CASxx operation code.

CLEAR and RESET Operation Codes

The CLEAR operation code sets variables in a window or a structure to their default values. If you specify a structure (record format, data structure or array) all fields in that structure are cleared in the order in which they are declared.

The RESET operation code sets variables in a window or a structure to their initial values. You can use data structure initialization to assign initial values to subfields, and then change the values during the running of the program, and use the RESET operation code to set the field values back to their initial values.

Data Initialization

Data is initialized with the INZ keyword on the definition specification. You can specify an initial value as a parameter on the INZ keyword, or specify the keyword without a parameter and use the default initial values. Default initial values for the various data types are described in Chapter 9, "Data Types and Data Formats." See Chapter 12, "Using Arrays and Tables" for information on initializing arrays.

Part 3. Specifications

This section describes the VisualAge RPG specifications:

- Information that is common to several specifications, such as keyword syntax and continuation rules, is described.
- Each specification is described in the order in which it must be entered in the program. Each specification description lists all the fields on the specification and explains all the possible entries.

Chapter 15. About VisualAge RPG Specifications

The VisualAge RPG language consists of a mixture of position-dependent and free-form code. A VisualAge RPG program is coded on a variety of specifications. Each specification has a specific set of functions.

There are three groups of source records that may be coded in a VisualAge RPG program: the main source section, the subprocedure section, and the program data section. The structure of the **main source section** depends on the resultant compilation target: component, NOMAIN DLL, or EXE. The **main source section** contains all of the global definitions for a module. For a component target, this section also includes the action and user subroutines. The layout of the main source section for each compilations target is shown in "Placement of Definitions and Scope" on page 256.

The **subprocedure section** contains specifications that define any subprocedures coded within a module. The **program data section** contains source records with data that is supplied at compile time.

The following illustration shows the types of source records that may be entered into each group and their order.

Note: The VisualAge RPG source program must be entered into the system in the order shown. Any of the specification types can be absent, but at least one must be present.

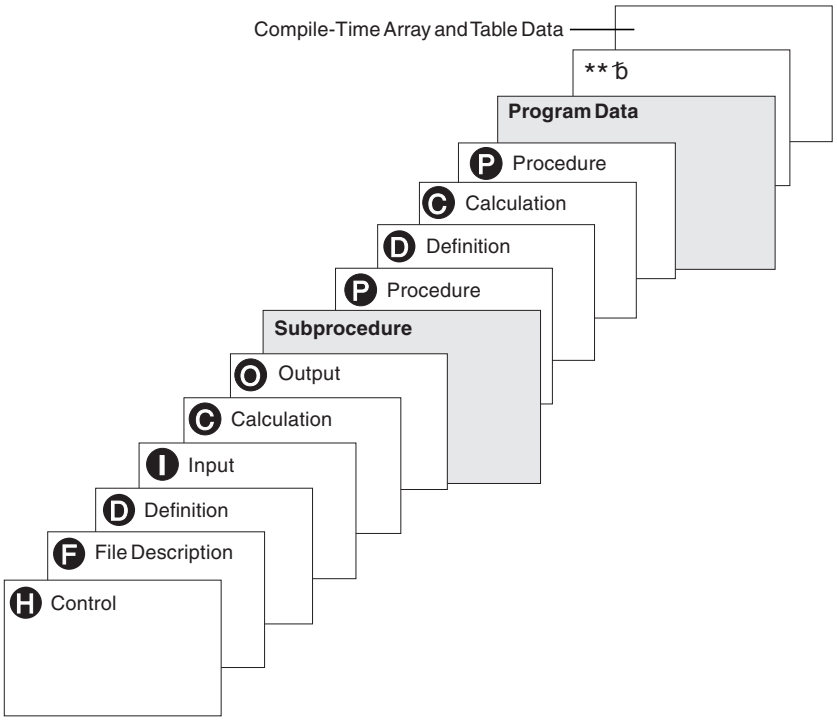


Figure 74. Order of the Types of Specifications in an VisualAge RPG Source Program

H Control (Header) specifications provide information about program

generation and running of the compiled program. Refer to Chapter 16, “Control Specifications” for a description of the entries on this specification.

F File description specifications define all files in the program. Refer to Chapter 17, “File Description Specifications” for a description of the entries on this specification.

D Definition specifications define items used in your program. Arrays, tables, data structures, subfields, constants, standalone fields, event attributes, prototypes and their parameters, and procedure interfaces and their parameters are defined on this specification. Refer to Chapter 18, “Definition Specifications” for a description of the entries on this specification.

I Input specifications describe records, and fields in the input files and indicate how the records and fields are used by the program. Refer to Chapter 19, “Input Specifications” for a description of the entries on this specification.

C Calculation specifications describe calculations to be done by the program and indicate the order in which they are done. Calculation specifications can control certain input and output operations. For component targets, this section includes action subroutines and standalone user subroutines. NOMAIN DLLs and EXEs do not have a calculation specifications section. Refer to Chapter 20, “Calculation Specifications” for a description of the entries on this specification. Chapter 23, “Operations” describes the operation codes that are coded on the Calculation specification.

O Output specifications describe the records and fields and indicate when they are to be written by the program. Refer to Chapter 21, “Output Specifications” for a description of the entries on this specification.

Subprocedure Specifications

P Procedure specifications describe the procedure-interface definition of a prototyped program or procedure. Refer to Chapter 22, “Procedure Specifications” for a description of the entries on this specification.

D Definition specifications define items used in the prototyped procedure. Procedure-interface definitions, entry parameters, and other local items are defined on this specification. Refer to Chapter 18, “Definition Specifications” for a description of the entries on this specification.

C Calculation specifications perform the logic of the prototyped procedure. Refer to Chapter 20, “Calculation Specifications” for a description of the entries on this specification.

Program Data

Source records with program data follow all source specifications. The first line of the data section must start with **. If desired, you can indicate the type of program data that follows the **, by specifying the CTDATA keyword. By associating the program data with this keyword, you can place the groups of program data in any order after the source records.

The first entry for each input record must begin in position 1. The entire record need not be filled with entries. Array elements with unused entries will be initialized with the default value.

For more information on entering compile-time array records, see “Rules for Array Source Records” on page 174.

The specifications which support keywords (Control, File Description, Definition, and Procedure) allow free format in the keyword fields. The Calculation specification allows free format with those operation codes which support extended-factor 2. Otherwise, entries are position specific. To represent this, each illustration of VisualAge RPG code is in listing format with a scale drawn across the top.

This reference contains a detailed description of the individual specifications. Each field and its possible entries are described. Chapter 23, “Operations,” on page 341 describes the operation codes that are coded on the Calculation specification, which is described in Chapter 20, “Calculation Specifications,” on page 311.

Common Entries

The following entries are common to all VisualAge RPG specifications:

- Positions 1-5 can be used for comments.
- Position 6 indicates the specification type. You can use the following letter codes:

Entry	Specification Type
H	Control
F	File description
D	Definition
I	Input
C	Calculation
O	Output
P	Procedure

- Comment Statements
  - Position 7 contains an asterisk (*). This denotes the line as a comment line regardless of any other entry on the specification. In a free-form calculation specification, you can use // for a comment. Any line on any fixed-form specification that begins with // is considered a comment by the compiler. The // can start in any position provided that positions 6 to the // characters contain blanks.
  - Positions 6 - 80 is blank
- Positions 7 - 80 are blank and position 6 contains a valid specification. This is a valid line, not a comment, and sequence rules are enforced.

Syntax of Keywords

Keywords may have no parameters, optional parameters, or required parameters. The syntax for keywords is as follows:

```
Keyword(parameter1 : parameter2)
```

where:

- Parameter(s) are enclosed in parentheses ( ).

  Note: Parentheses should not be specified if there are no parameters.
- Colons (:) are used to separate multiple parameters.

The following notational conventions are used to show which parameters are optional and which are required:

- Braces { } indicate optional parameters or optional elements of parameters.
- An ellipsis (...) indicates that the parameter can be repeated.
- A colon (:) separates parameters and indicates that more than one may be specified. All parameters separated by a colon are required unless they are enclosed in braces.
- A vertical bar (|) indicates that only one parameter may be specified for the keyword.
- A blank separating keyword parameters indicates that one or more of the parameters may be specified.

Note: Braces, ellipses, and vertical bars are not a part of the keyword syntax and should not be entered into your source.

Table 24. Examples of Keyword Notation

Notation	Example of Notation Used	Description	Example of Source Entered
braces {}	PRTCTL (data_struct {:*COMPAT})	Parameter data_struct is required and parameter *COMPAT is optional.	PRTCTL (data_struct1)
braces {}	TIME(format {separator})	Parameter format{separator} is required, but the {separator} part of the parameter is optional.	TIME(*HMS&)
colon (:)	RENAME(Ext_format :Int_format)	Parameters Ext_format and Int_format are required.	RENAME (nameE: nameI)
ellipsis (...)	IGNORE(recformat {:recformat...})	Parameter recformat is required and can be specified more than once.	IGNORE (recformat1: recformat2: recformat3)
vertical bar (|)	FLTDIV{(*NO | *YES)}	Specify *NO or *YES, or no parameter.	FLTDIV
blank	OPTIONS(*OMIT *VARSIZE *STRING *TRIM *RIGHTADJ)	One of *OMIT, *VARSIZE, *STRING, *TRIM, or *RIGHTADJ is required and more than one parameter can be optionally specified.	OPTIONS (*OMIT: *VARSIZE: *STRING: *TRIM: *RIGHTADJ)

Continuation Rules

The fields which may be continued are:

- The keywords field on the control specification
- The keywords field on the file description specification
- The keywords field on the definition specification
- The Extended-factor 2 field on the calculation specification
- The constant/editword field on the output specification
- The Name field on the definition or the procedure specification

General rules for continuation are as follows:

- The continuation line must be a valid line for the specification being continued (H, F, D, C, or O in position 6).
- No special characters should be used when continuing specifications across multiple lines, except when a literal or name must be split. For example, the following pairs are equivalent. In the first pair, the plus sign (+) is an operator, even when it appears at the end of a line. In the second pair, the plus sign is a continuation character.

```
C                   eval      x = a + b
C                   eval      x = a +
C                                 b

C                   eval      x = 'abc'
C                   eval      x = 'ab+
C                                  c'
```

- Only blank lines, empty specification lines, or comment lines are allowed between continued lines.
- The continuation can occur after a complete token. Tokens are:
  - Names (for example, keywords, file names, field names)
  - Parentheses
  - The separator character (:)
  - Expression operators
  - Built-in functions
  - Special words
  - Literals
- A continuation can also occur within a literal:
  - For character, date, time, and timestamp literals:
    - A hyphen (-) indicates continuation is in the first available position in the continued field
    - A plus (+) indicates continuation with the first nonblank character in or past the first position in the continued field
  - For graphic literals :
    - Either the hyphen (-) or plus (+) can be used to indicate a continuation.
  - For numeric literals:
    - No continuation character is used
    - A numeric literal continues with a numeric character or decimal point on the continuation line in the continued field
  - For hexadecimal and UCS-2 literals:
    - Either a hyphen (-) or a plus (+) can be used to indicate a continuation
    - The literal will be continued with the first nonblank character on the next line
- A continuation can also occur within a name in free-format entries
  - In the name entry for Definition and Procedure specifications. For more information on continuing names in the name entry, see "Definition and Procedure Specification Name Field" on page 220.
  - In the keywords entry for File and Definition specifications.
  - In the extended factor 2 entry of Calculation specifications.

You can split a qualified name at a period, as shown below:

```
C                   EVAL      dataStructureWithALongName.
C                               subfieldWithAnotherLongName = 5
```

If a name is not split at a period, code an ellipsis (...) at the end of the partial name, with no intervening blanks.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D                                     Keywords-cont++++++++++++++++++++++++++
* Define a 10 character field with a long name.
* The second definition is a pointer initialized to the address
* of the variable with the long name.
D QuiteLongFieldNameThatCannotAlwaysFitInOneLine...
D                 S             10A
D Ptr             S               *   inz(%addr(QuiteLongFieldName...
D                                             ThatCannotAlways...
D                                             FitInOneLine))
D ShorterName     S              5A

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++++
C                           Extended-factor2-++++++++++++++++++++++++++++++
* Use the long name in an expression
* Note that you can split the name wherever it is convenient.
C                   EVAL          QuiteLongFieldName...
C                                 ThatCannotAlwaysFitInOneLine = 'abc'

* You can split any name this way
C                   EVAL          P...
C                                 tr = %addr(Shorter...
C                                 Name)
```

Figure 75. Example

Control Specification Keyword Field

The rule for continuation on the control specification is that the specification continues on or past position 7 of the next control specification.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DATFMT(
H       *MDY&
H       )
```

Figure 76. Continuation on a Control Specification

File Description Specification Keyword Field

The rules for continuation on the file description specification are:

- The specification continues on or past position 44 of the next file description specification
- Positions 7-43 of the continuation line must be blank

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++
F                                     RECNO
F                                     (
F                                     *INU1
F                                     )
```

Figure 77. File description specification keyword field

Definition Specification Keyword Field

The rules for continuation on the definition specification are:

- The specification continues on or past position 44 of the next Definition specification depending on the continuation character specified
- Positions 7-43 of the continuation line must be blank.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++++
D                                   Keywords-cont+++++++++++++++++++++++++
 *
DMARY             C                 CONST(
D                                   'Mary had a little lamb, its -
D* Only a comment or a completely blank line is allowed in here
D                                   fleece was white as snow.'
D                                   )
D* Numeric literal, continues with the first non blank in/past position 44
D*
DNUMERIC          C                 12345
D                                     67
D* Graphic named constant
DGRAF             C                 G'AABBCCDD+
D                                    EEFFGG'
```

Figure 78. Definition specification keyword field example

Calculation Specification Extended-Factor 2

The rules for continuation on the calculation specification are:

- The specification continues on or past position 36 of the next calculation specification
- Positions 7-35 of the continuation line must be blank.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++++
C                             Extended-factor2-++++++++++++++++++++++++++++++
 *
C                   EVAL      MARY='Mary had a little lamb, its +
C* Only a comment or a completely blank line is allowed in here
C                               fleece was white as snow.'
C*
C*  Continuation of arithmetic expression, NOT a continuation
C*  character
C
C                   EVAL      A = (B*D)/ C +
C                             24
C*  The first use of '+' in this example is the concatenation
C*  operator.  The second use is the character literal continuation.
C                   EVAL      ERRMSG = NAME +
C                                      ' was not found +
C                                      in the file.'
```

Figure 79. Calculation specification Extended-Factor 2 Example

Free-Form Calculation Specification

The rules for continuation on a free-form calculation specification are:

- The free-form line can be continued on the next line. The statement continues until a semicolon is encountered.

  Example

```
/FREE
          time = hours * num_employees
                   + overtime_saved;
/END-FREE
```

Output Specification Constant/Editword Field

The rules for continuation on the Output specification are:

- The specification continues on or past position 53 of the next Output specification
- Positions 7-52 of the continuation line must be blank.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
 *
O.............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat+++
O                                          Continue Constant/editword+++

O                                       80 'Mary had a little lamb, its-
O* Only a comment or a completely blank line is allowed in here
O                                           fleece was white as snow.'
```

Figure 80. Output specification constant/editword field example

Definition and Procedure Specification Name Field

The rules for continuation of the name on the definition and procedure specifications are:

- Continuation rules apply for names longer than 15 characters. Any name (even one with 15 characters or fewer) can be continued on multiple lines by coding an ellipsis (...) at the end of the partial name.
- A name definition consists of the following parts:
  1. Zero or more continued name lines. Continued name lines are identified as having an ellipsis as the last non-blank characters in the entry. The name must begin within positions 7 - 21 and may end anywhere up to position 77 (with an ellipsis ending in position 80). There cannot be blanks between the start of the name and the ellipsis (...) characters. If any of these conditions is not true, the line is considered to be a main definition line.
  2. One main definition line containing name, definition attributes, and keywords. If a continued name line is coded, the name entry of the main definition line may be left blank.
  3. Zero or more keyword continuation lines.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D                                   Keywords-cont++++++++++++++++++++++++++
D* Long name without continued name lines:
D RatherLongName  S             10A
D* Long name using 1 continued name line:
D NameThatIsEvenLonger...
D                 C                   'This is the constant -
D                                     that the name represents.'
D* Long name using 1 continued name line:
D NameThatIsSoLongItMustBe...
D   Continued     S             10A
D* Compile-time arrays may have long names:
D CompileTimeArrayContainingDataRepresentingTheNamesOfTheMonthsOf...
D  TheYearInGermanLanguage...
D                 S             20A   DIM(12) CTDATA PERRCD(1)
D* Long name using 3 continued name lines:
D ThisNameIsSoMuchLongerThanThe...
D    PreviousNamesThatItMustBe...
D    ContinuedOnSeveralSpecs...
D                 PR            10A
D parm_1                        10A   VALUE
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C* Long names defined on calc spec:
C     LongTagName   TAG
C     *LIKE         DEFINE    RatherLongNameQuiteLongName    +5
PName++++++++++..B..................Keywords++++++++++++++++++++++++++++++
PContinuedName++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P* Long name specified on Procedure spec:
P ThisNameIsSoMuchLongerThanThe...
P    PreviousNamesThatItMustBe...
P    ContinuedOnSeveralSpecs...
P                 B
D ThisNameIsSoMuchLongerThanThe...
D    PreviousNamesThatItMustBe...
D    ContinuedOnSeveralSpecs...
D                 PI            10A
D parm_1                        10A   VALUE
D* Body of the Procedure
D*
P ThisNameIsSoMuchLongerThanThe...
P    PreviousNamesThatItMustBe...
P    ContinuedOnSeveralSpecs...
P                 E
```

Figure 81. Defining long names in RPG

Chapter 16. Control Specifications

The control specification statement, identified by an H in column 6, provides information about generating and running programs. This information is provided to the compiler by means of a control specification included in your source. If no control specification is included, the control specification keywords are assigned their default values.

See the description of the individual entries for their default values.

The control specification keywords apply at the modular level. This means that if there is more than one procedure coded in a module, the values specified in the control specification apply to all procedures.

Control Specification Statement

The control specification consists solely of keywords. The keywords can be placed anywhere between positions 7 and 80. Positions 81-100 can be used for comments.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 82. Control Specification Layout

The following is an example of a control specification:

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H CURSYM('$') DATEDIT(*MDY) DATFMT(*MDY/)
H DECEDIT('.') TIMFMT(*ISO)
```

Position 6 (Form Type)

An H must appear in position 6 to identify this line as the control specification.

Positions 7-80 (Keywords)

The control specification keywords are used to determine how the program deals with devices and how certain types of information are represented.

Syntax of Keywords

Control-specification keywords may have no parameters, optional parameters, or required parameters. The syntax for keywords is as follows:

```
Keyword(parameter1 : parameter2)
```

where:
- Parameter(s) are enclosed in parentheses ( ).

  Note: Do not specify parentheses if there are no parameters.
- Colons (:) are used to separate multiple parameters.

The following notational conventions are used to show which parameters are optional and which are required:
- Braces { } indicate optional parameters or optional elements of parameters.
- An ellipsis (...) indicates that the parameter can be repeated.
- A colon (:) separates parameters and indicates that more than one may be specified. All parameters separated by a colon are required unless they are enclosed in braces.
- A vertical bar (|) indicates that only one parameter may be specified for the keyword.
- A blank separating keyword parameters indicates that one or more of the parameters may be specified.

Note: Braces, ellipses, and vertical bars are not a part of the keyword syntax and should not be entered into your source.

If additional space is required for control-specification keywords, the keyword field can be continued on subsequent lines. See "Control Specification Statement" on page 223 and "Control Specification Keyword Field" on page 217.

ALWNULL(*NO | *INPUTONLY | *USRCTL)

The ALWNULL keyword specifies how you will use records containing null-capable fields from externally described database files.

If ALWNULL(*NO) is specified, then you cannot process records with null-value fields from externally described files. If you attempt to retrieve a record containing null values, no data in the record will be accessible and a data-mapping error will occur.

If ALWNULL(*INPUTONLY) is specified, then you can successfully read records with null-capable fields containing null values from externally described input-only database files. When a record containing null values is retrieved, no data-mapping errors will occur and the database default values are placed into any fields that contain null values. However, you cannot do any of the following:
- Use null-capable key fields
- Create or update records containing null-capable fields
- Determine whether a null-capable field is actually null while the program is running
- Set a null-capable field to be null.

If ALWNULL(*USRCTL) is specified, then you can read, write, and update records with null values from externally described database files. Records with null keys can be retrieved using keyed operations. You can determine whether a null-capable

field is actually null, and you can set a null-capable field to be null for output or update. You are responsible for ensuring that fields containing null values are used correctly.

If the ALWNULL keyword is not specified, then the value specified on the command is used.

For more information, see "Database Null Value Support" on page 137

CACHE(*YES | *NO)

The CACHE keyword specifies that the remote file descriptions stored on the workstation in the cache folder is to be used by the application. The first time that the CACHE(*YES) option is used, a list of remote file descriptions will be created. Each subsequent time, the compile process will access this information instead of accessing the server files.

CACHEREFRESH(*YES | *NO)

The CACHEREFRESH keyword specifies that the remote file descriptions in the cache folder is to be updated before the compile process. If you specify CACHE(*NO), the existing remote file descriptions are used.

CCSID(*GRAPH : parameter | *UCS2 : number | *MAPCP : 932)

This keyword sets the default graphic (*GRAPH) and UCS-2 (*UCS2) CCSIDs for the module. These defaults are used for compile-time data, program-described input and output fields, and data definitions that do not have the CCSID keyword coded.

CCSID(*GRAPH : *IGNORE | *SRC | number)
Sets the default graphic CCSID for the module. The possible values are:

***IGNORE**
This is the default. No conversions are allowed between graphic and UCS-2 fields in the module. The %GRAPH built-in function cannot be used.

***SRC**
The graphic CCSID associated with the CCSID of the source file will be used.

number
A graphic CCSID.

CCSID(*UCS2 : number)
Sets the default UCS-2 CCSID for the module. If this keyword is not specified, the default UCS-2 CCSID is 13488.

number must be a UCS-2 CCSID. Valid CCSIDs are 13844 and 17584 (which inlcudes the Euro).

CCSID(*MAPCP : 932)
For remote file opens and program calls, maps the Japanese code page 932 to CCSID 943.

If CCSID(*GRAPH : *SRC) or CCSID(*GRAPH : number) is specified:
- Graphic and UCS-2 fields in externally-described data structures will use the CCSID in the external file.

- Program-described graphic or UCS-2 fields will default to the graphic or UCS-2 CCSID of the module, respectively. This specification can be overridden by using the CCSID(number) keyword on the definition of the field. (See "CCSID(number | *DFT)" on page 267.)
- Program-described graphic or UCS-2 input and output fields and keys are assumed to have the module's default CCSID.

COPYNEST(number)

The COPYNEST keyword specifies the maximum depth to which nesting can occur for /COPY directives. The depth must be greater than or equal to 1 and less than or equal to 2048. The default depth is 32.

COPYRIGHT('copyright string')

The COPYRIGHT keyword provides copyright information. The copyright string is a character literal with a maximum length of 256. The literal may be continued on a continuation specification. (See "Continuation Rules" on page 215 for rules on using continuation lines.) If the COPYRIGHT keyword is not specified, copyright information is not added to the created module or program.

CURSYM('sym')

The CURSYM keyword specifies a character used as a currency symbol in editing. The symbol must be a single character enclosed in quotes. Any character in the VisualAge RPG character set may be used. (See Chapter 1, "Symbolic Names and Reserved Words," on page 3.) The following characters are exceptions:

0 (zero)	* (asterisk)	, (comma)
& (ampersand)	. (period)	– (minus sign)
C (letter C)	R (letter R)	Blank

If the keyword is not specified, the dollar sign ($) is the default for the currency symbol.

CVTOEM(*YES | *NO)

The CVTOEM keyword specifies that OEM conversion should be used when I/O is performed to local files. If you specify CVTOEM(*NO), no OEM conversion is done.

CVTOPT(*{NO}VARCHAR *{NO}VARGRAPHIC)

The CVTOPT keyword is used to determine how the VARPG compiler handles variable-length data types that are retrieved from externally described database files.

You can specify any or all of the data types in any order. However, if a data type is specified, the *NOxxxx parameter for the same data type cannot be used, and vice versa. For example, if you specify *VARCHAR you cannot specify *NOVARCHAR, and vice versa. Separate the parameters with a colon. A parameter cannot be specified more than once.

Note: If the keyword CVTOPT does not contain a member from a pair, then the value specified on the command for this particular data type will be used. For example, if the keyword CVTOPT(*NOVARCHAR) is specified on the

Control specification, then for the pair (*VARGRAPHIC, *NOVARGRAPHIC), whatever was specified implicitly or explicitly on the command will be used.

If *VARCHAR is specified, then variable-length character data types are declared as fixed-length character fields.

If *NOVARCHAR is specified, then variable-length character data types are not converted.

If *VARGRAPHIC is specified, then variable-length double-byte character set (DBCS) graphic data types are declared as fixed-length character fields.

If *NOVARGRAPHIC is specified, then variable-length double-byte character set (DBCS) graphic data types are not converted.

If the CVTOPT keyword is not specified, then the values specified on the command are used.

DATEDIT(fmt{separator})

The DATEDIT keyword specifies the format of numeric fields when using the Y edit code. The separator character is optional. The value (fmt) can be *DMY, *MDY, or *YMD. The default separator is /. A separator character of & (ampersand) may be used to specify a blank separator.

DATFMT(fmt{separator})

The DATFMT keyword specifies the internal date format for date literals and the default format for date fields within the program. You can specify a different internal date format for a particular field by specifying the format with the DATFMT keywork on the definition specification for that field.

The default is *ISO format. For more information on internal formats, see "Internal and External Formats" on page 103

Table 25 describes the various date formats and their separators.

Table 25. External Date Formats for Date Data Type

RPG name	Description	Format (Default Separator)	Valid Separators	Length	Example
*MDY	Month/Day/Year	mm/dd/yy	/ - . , '&'	8	01/15/91
*DMY	Day/Month/Year	dd/mm/yy	/ - . , '&'	8	15/01/91
*YMD	Year/Month/Day	yy/mm/dd	/ - . , '&'	8	91/01/15
*JUL	Julian	yy/ddd	/ - . , '&'	6	91/015
*ISO	International Standards Organization	yyyy-mm-dd	-	10	1991-01-15
*USA	IBM USA Standard	mm/dd/yyyy	/	10	01/15/1991
*EUR	IBM European Standard	dd.mm.yyyy	.	10	15.01.1991
*JIS	Japanese Industrial Standard Christian Era	yyyy-mm-dd	-	10	1991-01-15

DEBUG{(*NO | *YES)}

The DEBUG keyword determines whether debug information is generated.

If this keyword is not specified or specified with *NO, no debug information is generated.

DECEDIT('value')

This keyword specifies the character used as the decimal point for edited decimal numbers. Leading zeros are printed when the absolute value of the number is less than 1. The default value is '.' (period).

The possible values are:

'.' Decimal point is period; leading zero not printed (.123)

',' Decimal point is comma; leading zero not printed (,123)

'0.' Decimal point is period; leading zero printed (0.123)

'0,' Decimal point is comma; leading zero printed (0,123)

DECPREC(30|31)

Keyword DECPREC is used to specify the decimal precision of decimal (packed, zoned, or binary) intermediate values in arithmetic operations in expressions. Decimal intermediate values are always maintained in the proper precision, but this keyword affects how decimal values are presented when used in certain operations, such as %EDITC and %EDITW.

DECPREC(30)
The default decimal precicion. It indicates that the maximum precision of decimal values when used in %EDITC and %EDITW operations is 30 digits. However, if at least one operand in the expression is a 31 digit decimal variable, the precision of the expression is 31 digits regardless of the DECPREC value.

DECPREC(31)
This alternate decimal precision indicates that 31 digits are always presented in %EDITC and %EDITW operations.

EXE

The EXE keyword indicates that this is a module consisting of a main procedure and subprocedures. All subroutines (BEGSR) must be local to a procedure. The EXE must contain a procedure whose name matches the name of the source file. This will be the main entry point for the EXE, that is, the main procedure.

For EXE modules, consider the following:
- No GUI operation codes are allowed in the source. This includes START, STOP, SETATR, GETATR, %SETATR, %GETATR, SHOWWIN, CLSWIN, and READS. DSPLY can be used.
- *INZSR and *TERMSR are not permitted.
- *ENTRY parameters are not permitted.

If there are entry parameters, they are specified on the parameter definition for the main procedure, and they must be passed in by value, that is, the VALUE keyword must be specified for each parameter.
- The EXPORT keyword is not allowed on the Begin P specification.
- The return value for the main procedure must be defined as a binary or integer of precision zero(0).

EXPROPTS(*MAXDIGITS | *RESDECPOS)

The EXPROPTS (expression options) keyword specifies the type of precision rules to be used for an entire program. If not specified or specified with *MAXDIGITS, the default precision rules apply. If EXPROPTS is specified, with *RESDECPOS, the "Result Decimal Position" precision rules apply and force intermediate results in expressions to have no fewer decimal positions than the result.

Note: Operation code extenders R and M are the same as EXPROPTS(*RESDECPOS) and EXPROPTS(*MAXDIGITS) respectively, but for single free-form expressions.

EXTBININT{(*NO | *YES)}

The EXTBININT keyword is used to process externally described fields with binary external format and zero decimal positions as if they had an external integer format. If not specified or specified with *NO, then an externally described binary field is processed with an external binary format. If EXTBININT is specified, optionally with *YES, then an externally described field is processed as follows:

DDS Definition	RPG external format
B(n,0) where 1 ≤ n ≤ 4	I(5)
B(n,0) where 5 ≤ n ≤ 9	I(10)

By specifying the EXTBININT keyword, your program can make use of the full range of DDS binary values available. (The range of DDS binary values is the same as for signed integers: -32768 to 32767 for a 5-digit field or -2147483648 to 2147483647 for a 10-digit field.)

Note: When the keyword EXTBININT is specified, any externally described subfields that are binary with zero decimal positions will be defined with an *internal* integer format.

FLTDIV{(*NO | *YES)}

The FLTDIV keyword indicates that all divide operations within expressions are computed in floating point and return a value of type float. If not specified or specified with *NO, then divide operations are performed in packed-decimal format (unless one of the two operands is already in float format).

If FLTDIV is specified, optionally with *YES, then all divide operations are performed in float format (guaranteeing that the result always has 15 digits of precision).

GENLVL(number)

The GENLVL keyword controls the creation of the object. The object is created if all errors encountered during compilation have a severity level less than or equal to the generation severity level specified. The value must be between 0 and 20 inclusive. For errors greater than severity 20, the object will not be created.

If the GENLVL keyword is not specified, then the value specified on the command is used.

INDENT(*NONE | 'character-value')

The INDENT keyword specifies whether structured operations should be indented in the source listing for enhanced readability. It also specifies the characters that are used to mark the structured operation clauses.

If *NONE is specified, structured operations will not be indented in the source listing.

If character-value is specified, the source listing is indented for structured operation clauses. Alignment of statements and clauses are marked using the characters you choose. You can choose any character literal up to 2 characters in length.

Note: The indentation may not appear as expected if there are errors in the source.

If the INDENT keyword is not specified, then the value specified on the command is used.

INTPREC(10 | 20)

The INTPREC keyword is used to specify the decimal precision of integer and unsigned intermediate values in binary arithmetic operations in expressions. Integer and unsigned intermediate values are always maintained in 8-byte format. This keyword affects only the way integer and unsigned intermediate values are converted to decimal format when used in binary arithmetic operations (+, -, *, /).

INTPREC(10), the default, indicates a decimal precision of 10 digits for integer and unsigned operations. However, if at least one operand in the expression is an 8-byte integer or unsigned field, the result of the expression has a decimal precision of 20 digits regardless of the INTPREC value.

INTPREC(20) indicates that the decimal precision of integer and unsigned operations is 20 digits.

LIBLIST('filename1 filename2 ... filenamen')

The LIBLIST keyword specifies the list of library files to be used when linking the application. Each file name must be separated by a blank and the list must be enclosed by single quotation marks. If a file name contains blanks, its name must be enclosed by double quotation marks.

NOMAIN

The NOMAIN keyword indicates that there are no action or standalone user subroutines in the module. A **NOMAIN module** contains only subprocedures. The resulting compilation target is a DLL that can be used by other applications.

For NOMAIN DLLs, the following should be considered:

- The DLL must consist of procedures only. All subroutines (BEGSR) must be local to a procedure.
- No GUI operation codes allowed in the source. These include START, STOP, SETATR, GETATR, %SETATR,%GETATR;, SHOWWIN, CLSWIN, and READS. DSPLY can be used. However, if the procedure containing it is called from a VisualAge RPGDLL, then the DSPLY operation code does nothing.
- *INZSR; and *TERMSR; are not permitted.
- *ENTRY; parameters are not permitted.

See *Programming with VisualAge RPG* for information on coding and calling multiple procedures.

OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP *{NO}INHERITSIGNON)

The OPTION keyword specifies the options to use when the source member is compiled.

You can specify any or all of the options in any order. However, if a compile option is specified, the *NOxxxx parameter for the same compile option cannot be used, and vice versa. For example, if you specify *XREF, you cannot also specify *NOXREF, and vice versa. Separate the options with a colon. You cannot specify an option more than once.

Note: If the keyword OPTION does not contain a member from a pair, then the value specified on the command for this particular option will be used. For example, if the keyword OPTION(*XREF : *NOGEN : *NOSECLVL : *SHOWCPY) is specified on the Control specification, then for the pairs, (*EXT, *NOEXT), (*EXPDDS, *NOEXPDDS) and (*SHOWSKP, *NOSHOWSKP), whatever was specified implicitly or explicitly on the command will be used.

If *XREF is specified, a cross-reference listing is produced (when appropriate) for the source member. *NOXREF indicates that a cross-reference listing is not produced.

If *GEN is specified, a program object is created if the highest severity level returned by the compiler does not exceed the severity specified in the GENLVL option. *NOGEN does not create an object.

If *SECLVL is specified, second-level message text is printed on the line following the first-level message text in the Message Summary section. *NOSECLVL does not print second-level message text on the line following the first-level message text.

If *SHOWCPY is specified, the compiler listing shows source records of members included by the /COPY compiler directive. *NOSHOWCPY does not show source records of members included by the /COPY compiler directive.

If *EXPDDS is specified, the expansion of externally described files in the listing and key field information is displayed. *NOEXPDDS does not show the expansion of externally described files in the listing or key field information.

If *EXT is specified, the external procedures and fields referenced during the compile are included on the listing. *NOEXT does not show the list of external procedures and fields referenced during compile on the listing.

If *SHOWSKP is specified, then all statements in the source part of the listing are displayed, regardless of whether or not the compiler has skipped them. *NOSHOWSKP does not show skipped statements in the source part of the listing. The compiler skips statements as a result of /IF, /ELSEIF, or /ELSE directives.

If *INHERITSIGNON is specified, the calling application's server signon information is used by the called program. This avoids the user ID/password prompting when data or programs are accessed on the remote server.

If the OPTION keyword is not specified, then the values specified on the command are used.

SIGNON(*CLEARUSERID *HIDEPWSAVE *INHERIT)

The SIGNON keyword specifies options to use when signing on to a remote server to access files or call programs.

You can specify any or all of the options in any order. Separate the options with a colon. You cannot specify an option more than once.

If *CLEARUSERID is specified, no initial value will be shown in the user ID field of the remote server signon prompt dialog box, if the signon prompt is needed. (This option does not affect the Change Password prompt shown when an expired password is used to signon.)

The default behaviour is to show the last-used user ID.

If *HIDEPWSAVE is specified, the password save option will not be shown on the Change Password dialog box which appears when the server indicates a signon attempt used an expired password.

Note: If a VARPG application has a saved version of the password, and the actual password has changed, subsequent application signon attempts will report invalid password errors and prompt for the correct password, until the saved password entry is corrected or cleared.

If *INHERIT is specified in a VARPG application called by another, the calling application's server signon information is used by the called program. This avoids the user ID and password prompting when data or programs are accessed on the remote server.

Do not specify both SIGNON(*INHERIT) and OPTION(*INHERITSIGNON), which mean the same.

SQLBINDFILE('filename')

The SQLBINDFILE keyword specifies that an SQL bind file be created. You can optionally specify a fully qualified bind file name enclosed in single quotation marks. The name can be up to 8 characters long.

A bind file allows the application to defer binding to a database until a later time and allows an application to access many databases. This is done using the SQLBIND command before the application runs.

No package file is generated unless you specify the SQLPACKAGENAME keyword. Applications can be built with binding enabled, that is, with the SQLPACKAGENAME keyword specified, or with binding deferred (no package name). Building with binding enabled generates a package file and stores it in the database. Building with binding deferred extracts the data needed to create the package from the source file and stores this information in a bind file.

SQLDBBLOCKING(*YES | *NO)

The SQLDBBLOCKING keyword specifies whether blocking is done on any cursors. Specify SQLDBBLOCKING(*YES) to perform record blocking on any cursors.

When you use record blocking and specify SQLISOLATIONLVL(*RR), a read-only cursor isolation level, Database Manager at the database server returns a block of

rows to the database client in one network transmission. These rows are retrieved one at a time from the database client when Database Manager processes a FETCH request. When all rows in the block have been fetched, Database Manager at the database client sends another request to the remote database, until all output rows have been retrieved.

Record blocking can lead to results that are not entirely consistent with the database when used in combination with the cursor stability, SQLISOLATIONLVL(*CS), or uncommitted read, SQLISOLATIONLVL(*UR), isolation levels. With cursor stability and uncommitted read, the row being retrieved by the application from the block is not locked at the remote database. Therefore, another application may be updating the row in the database while your application is reading the row from the block. Specifying the repeatable read isolation level locks all accessed rows in the database until the unit of work is complete, but restricts updates by other processes.

Specify SQLDBBLOCKING(*NO) if you do not want blocking done on any cursors. When a SELECT statement returns multiple rows, the application must declare a cursor and use the FETCH statement to retrieve the rows one at a time. With a remote database, this means that each request and each reply travel across the network. With a large number of rows, this leads to a significant increase in network traffic.

SQLDBNAME('Dbname')

The SQLDBNAME keyword specifies the name of the DB2 database referenced by imbedded SQL statements in your application. The name must be enclosed by single quotation marks and can be up to 8 characters long.

SQLDTFMT(*EUR | *ISO | *USA | *JIS)

The SQLDTFMT keyword specifies the date and time format used in your application. The possible values are:

***EUR** IBM European Standard format.

***ISO** International Standards Organization format.

***USA** IBM USA Standard format.

***JIS** Japanese Industrial Standard Christian Era format.

SQLISOLATIONLVL(*RR | *CS | *UR)

The SQLISOLATIONLVL keyword specifies how SQL database records will be read by your application. The possible values are:

***RR** Repeatable read keeps a lock on all rows accessed by the application since the last commit point. If the application reads the same row again, the values will not have changed. The effect of the *RR isolation level is that one application can prevent other applications or users from changing tables. As a result, overall concurrency may decrease.

Specify repeatable read only if the application requires row locking; otherwise, cursor stability, *CS, is preferable.

***CS** Cursor stability, *CS, holds a row lock only while the cursor is positioned on that row. When the cursor moves to another row, the lock is released. If the data is changed, however, the lock must be held until the data is committed. Cursor stability applies only to data that is read. All changed data remains locked until either a COMMIT or ROLLBACK is processed.

Specify cursor stability if a given row will be accessed only once during the life of the transaction. In this way, the lock has the least impact on concurrent applications and users.

***UR** Uncommitted read, *UR, views rows without waiting for locks. Uncommitted read applies to FETCH and SELECT INTO operations. For other operations, the *UR choice performs the same as *CS, cursor stability. An application using this level reads and returns all rows, even if they contain uncommitted changes made by other applications. Because this isolation level does not wait for concurrency locks, overall performance may increase.

SQLPACKAGENAME('package.txt')

The SQLPACKAGENAME keyword specifies that a package file be created containing the executable SQL statements. You can optionally specify a fully qualified package name enclosed in single quotation marks. The name can be up to 8 characters long.

A Database Manager application uses one package file for every built source file used to build the application. Each package is a separate entity and has no relationship to any other packages used by the same or other applications. Packages are created by running the precompiler against a source file with binding enabled or by running the binder (SQLBIND command) against one or more DB2 names.

SQLPASSWORD('password')

The SQLPASSWORD keyword specifies the password of the user ID accessing the SQL database. The password must be enclosed by single quotation marks

SQLUSERID('userid')

The SQLUSERID keyword specifies the user ID connecting to the SQL database. The user ID must be enclosed by single quotation marks

TIMFMT(fmt{separator})

The TIMFMT keyword specifies the internal format of time literals and the default format for time fields in the program. You can specify a different internal time format for a particular field by specifying the format with the TIMFMT keyword on the definition specification for that field.

The default is *ISO. For more information on the internal formats, see "Internal and External Formats" on page 103

Table 26 shows the time formats supported and their separators:

Table 26. External Time Formats for Time Data Type

RPG name	Description	Format (Default Separator)	Valid Separators	Length	Example
*HMS	Hours:Minutes:Seconds	hh:mm:ss	: . , &	8	14:00:00
*ISO	International Standards Organization	hh.mm.ss	.	8	14.00.00
*USA	IBM USA Standard. AM and PM can be any mix of upper and lower case.	hh:mm AM or hh:mm PM	:	8	02:00 PM
*EUR	IBM European Standard	hh.mm.ss	.	8	14.00.00
*JIS	Japanese Industrial Standard Christian Era	hh:mm:ss	:	8	14:00:00

TRUNCNBR(*YES | *NO)

The TRUNCNBR keyword specifies if the truncated value is moved to the result field or if an error is generated when numeric overflow occurs while running the object.

Note: The TRUNCNBR option does not apply to calculations performed within expressions. (Expressions are found in the Extended-Factor 2 field.) If overflow occurs for these calculations, an error will always occur.

If *YES is specified, numeric overflow is ignored and the truncated value is moved to the result field.

If *NO is specified, a run-time error is generated when numeric overflow is detected.

If the TRUNCNBR keyword is not specified, then the value specified on the command is used.

Chapter 17. File Description Specifications

File description specifications identify each file used by a program. Each file in a program must have a corresponding file description specification statement.

A file can be either program-described or externally-described. In program-described files, record and field descriptions are included within the program using input and output specifications. Externally-described files have their record and field descriptions defined externally on an iSeries server using DDS or SQL/400™ commands.

The following limitations apply for each program:
- There is no limit for the maximum number of files allowed
- DISK files:
  - DISK files can be either remote or local
  - Remote files must be externally described
  - Local files must be program described
- PRINTER files:
  - A maximum of eight PRINTER files are allowed
  - PRINTER files must be program described
- SPECIAL files:
  - SPECIAL files must be program described.

File Description Specification Statement

The general layout for the file description specification is as follows:
- The file description specification type (F) is entered in position 6
- The non-comment part of the specification extends from position 7 to position 80:
  - The fixed-format entries extend from positions 7 to 42
  - The keyword entries extend from positions 44 to 80
- The comments section of the specification extends from position 81 to position 100.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 83. File Description Specification Layout

File-Description Keyword Continuation Line

If additional space is required for keywords, the keywords field can be continued on subsequent lines as follows:
- position 6 of the continuation line must contain an F
- positions 7 to 43 of the continuation line must be blank
- the specification continues on or past position 44

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
F.....................................Keywords++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 84. File-Description Keyword Continuation Line Layout

Position 6 (Form Type)

An F must be entered in this position.

Positions 7-16 (File Name)

Entry	Explanation
A valid file name	Every file used in a program must have a unique name. The file name can be from 1 to 10 characters long, and must begin in position 7.

For an externally-described file, the file must exist at both compilation time and at run time. For a program-described file, the file needs to exist only at run time.

At run time:
- If you use the EXTFILE keyword, the EXTMBR keyword (remote OS/400 files only), or both, RPG will open the file named in these keywords.
- Otherwise, RPG will open the file named in position 7. This file (or an overridden file) must exist when the file is opened.
- For remote OS/400 files, if an OS/400 system override command has been used for the file that RPG opens, that override will take effect and the actual file opened will depend on the override. See the "EXTFILE(filename)" on page 246 keyword for more information about how overrides interact with this keyword.

When the files are opened at run time, they are opened in the reverse order to that specified in the file-description specifications. The device name defines the operations that can be processed on the associated file.

Program-Described File

For program-described files, the file name entered in positions 7 through 16 must also be entered on:
- Input specifications
- Output specifications or an output calculation operation line if the file is an output, update, or combined file, or if file addition is specified for the file
- Definition specifications if the file is a table or array file
- Calculation specifications if the file name is required for the operation code specified.

Externally-Described File

For externally described files, the file name entered in positions 7 through 16 is the name used to locate the record descriptions for the file. The following rules apply to externally described files:
- Input and output specifications for externally described files are optional. They are required only if you are adding VisualAge RPG functions, such as record identifying indicators, to the external description retrieved.
- When an external description is retrieved, the record definition can be referred to by its record format name on the input, output, or calculation specifications.
- A record format name must be a unique symbolic name.
- An externally-described logical file with two record formats of the same name is not allowed.

Position 17 (File Type)

Entry	Explanation
I	An Input file can be either a local or remote DISK file

O An Output file can be either a local or remote DISK file

U An Update file can be either a local or remote DISK file

C A Combined (input/output) file must be a remote DISK file

Input Files

A program reads information from an input file. The input file can contain data records, arrays, or tables.

Output Files

An output file is a file to which information is written.

Update Files

An update file is an input file whose records can be read and updated. Updating alters the data in one or more fields of any record contained in the file and writes that record back to the same file from which it was read. If records are to be deleted, the file must be specified as an update file.

Combined Files

A combined file is both an input file and an output file. When a combined file is processed, the output record contains only the data represented by the fields in the output record. This differs from an update file, where the output record contains the input record modified by the fields in the output record.

A combined file is valid for a SPECIAL file and a DISK file if position 18 contains T (an array or table replacement file).

Position 18 (File Designation)

Entry	Explanation
Blank	Output file
T	Array or table file
F	Full procedural file

Array or Table File

Array and table files specified by a T in position 18 are loaded at program initialization time. The array or table file can be input or combined. Leave this entry blank for array or table output files. You cannot specify SPECIAL as the device for array and table input files. You cannot specify an externally described file as an array or table file.

If T is specified in position 18, you can specify a file type of combined (C in position 17) for a DISK file. A file type of combined allows an array or table file to be read from or written to the same file (an array or table replacement file) or to a different file. In addition, the file name in positions 7–16 must also be specified as the parameter to the TOFILE keyword on the definition specification.

Full Procedural File

For a full procedural file, input is controlled by calculation operations. File operation codes such as CHAIN or READ are used to do input functions.

Position 19 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Position 20 (File Addition)

Position 20 indicates whether records are to be added to an input or update file. For output files, this entry is ignored.

Entry	Explanation
Blank	No records can be added to an input or update file (I or U in position 17).
A	Records are added to an input or update file when positions 18 through 20 of the output record specifications for the file contain ″ADD″, or when the WRITE operation code is used in the calculation specification.

See Table 27 on page 241 for the relationship between position 17 and position 20 of the file-description specifications and positions 18 through 20 of the output specifications.

Table 27. Processing Functions for Files

	Specification		
	File Description		Output
Function	**Position 17**	**Position 20**	**Positions 18-20**
Create new file[1] or Add records to existing file	O O	Blank A	Blank ADD
Process file	I	Blank	Blank
Process file and add records to the existing file	I	A	ADD
Process file and update the records (update or delete)	U	Blank	Blank
Process file and add new records to an existing file	U	A	ADD
Process file and delete an existing record from the file	U	Blank	DEL
: [1]Within RPG, the term *create a new file* means to add records to a newly created file. Thus, the first two entries in this table perform the identical function. Both are listed to show that there are two ways to specify that function.			

Position 21 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Position 22 (File Format)

Entry	Explanation
F	Program described file
E	Externally described file

An F in position 22 indicates that the records for the file are described within the program on input/output specifications (except for array/table files). PRINTER files and SPECIAL files must be program described. Local DISK files must be program described.

An E in position 22 indicates that the record descriptions for the file are external to the VisualAge RPG source program. The compiler obtains these descriptions at compilation time and includes them in the source program. Remote DISK files must be externally described.

Positions 23-27 (Record Length)

Use positions 23 through 27 to indicate the length of the logical records contained in a program described file. The maximum record size that can be specified is 32766; however, record-size constraints of any device may override this value. For PRINTER files, specify a record length which does not exceed the number of columns of printer output. This entry must be blank for externally described files.

Position 28 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Positions 29-33 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Position 34 (Record Address Type)

Entry	Explanation
Blank	Relative record numbers are used to process the file. Records are read consecutively.
K	Key values are used to process the file. This entry is valid only for externally described files.

Blank = Non-keyed Processing

A blank indicates that the file is processed without the use of keys.

A file processed without keys can be processed consecutively or randomly by relative-record number.

Input processing by relative-record number is determined by a blank in position 34 and by the use of the CHAIN, SETLL, or SETGT operation code. Output processing by relative-record number is determined by a blank in position 34 and by the use of the RECNO keyword on the file description specifications.

Key

A K entry indicates that the externally described file is processed on the assumption that the access path is built on key values. If the processing is random, key values are used to identify the records.

If this position is blank for a keyed file, the records are retrieved in arrival sequence.

Position 35 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Positions 36-42 (Device)

Entry	Explanation
PRINTER	File is a printer file, with control characters that can be sent to a printer.
DISK	File is a disk file. Sequential and random read/write processing is available for remote files. Sequential and relative record processing is available for local files.
SPECIAL	This is a special file. Input or output is on a device that is accessed by user-supplied code that is linked in to the VisualAge RPG application. The name of the user-supplied code module must be

specified as the parameter for the PROCNAME keyword. A parameter list is created for use with this program, including an option code parameter and a status code parameter. The file must be a fixed unblocked format. See "PLIST(Plist_name)" on page 248 and "PROCNAME(proc_name)" on page 250 for more information.

Use positions 36 through 42 to specify the device name to be associated with the file. The device name defines the functions that can be done on the associated file. Certain functions are valid only for a specific device name.

Position 43 (Reserved)

Position 43 must be blank.

Positions 44-80 (Keywords)

Positions 44 to 80 are provided for file description specification keywords. Keywords are used to provide additional information about the file being defined.

File-description specification keywords may have no parameters, optional parameters, or required parameters. The syntax for keywords is as follows:

```
Keyword(parameter1 : parameter2)
```

where:
- Parameter(s) are enclosed in parentheses ( ).

  Note: Do not specify parentheses if there are no parameters.
- Colons (:) are used to separate multiple parameters.

The following notational conventions are used to show which parameters are optional and which are required:
- Braces { } indicate optional parameters or optional elements of parameters.
- An ellipsis (...) indicates that the parameter can be repeated.
- A colon (:) separates parameters and indicates that more than one may be specified. All parameters separated by a colon are required unless they are enclosed in braces.
- A vertical bar (|) indicates that only one parameter may be specified for the keyword.
- A blank separating keyword parameters indicates that one or more of the parameters may be specified.

Note: Braces, ellipses, and vertical bars are not a part of the keyword syntax and should not be entered into your source.

If additional space is required for keywords, the keyword field can be continued on subsequent lines. See "File-Description Keyword Continuation Line" on page 237 and "File Description Specification Keyword Field" on page 217.

The following table summarizes which keywords apply to externally-described files and which keywords apply to program-described files.

Keyword	Program-described	Externally-described
BLOCK		Y
COMMIT{(rpg_name)}		Y
CVTHEX		Y

Keyword	Program-described	Externally-described
DATFMT(format{separator})	Y	Y
DEVMODE(name)	Y	
EOFMARK(*NONE)	Y	
EXTFILE(fname)	Y	
EXTMBR(membername)		Y
FORMLEN(number)	Y	
IGNORE(recformat{:recformat...})		Y
INCLUDE(recformat{:recformat...})		Y
INFDS(DSname)	Y	Y
INFSR(SUBRname)	Y	Y
PLIST(Plist_name)	Y	Y
PREFIX(prefix_name)		Y
PROCNAME(proc_name)	Y	
PRTCTL(data_struct{:*COMPAT})	Y	
PRTFMT(*SYS | *TEXT)	Y	
RCDLEN(fieldname)	Y	
RECNO(fieldname)	Y	Y
REMOTE		Y
RENAME(Ext_format:Int_format)		Y
TIMFMT(format{separator})	Y	Y
USROPN	Y	Y

BLOCK(*YES|*NO)

The BLOCK keyword controls the blocking of records associated with the file. The keyword is valid only for DISK files.

If this keyword is not specified, the VARPG compiler unblocks input records and blocks output records to improve runtime performance in DISK files when the following conditions are met:

1. The file is externally described and has only one record format.
2. The RECNO keyword is not used in the file description specification.
3. One of the following is true:
   a. The file is an output file.
   b. If the file is a combined file, then it is an array or table file.
   c. The file is an input-only file and none of the following operations are used on the file: READE, READPE, SETGT, SETLL, and CHAIN. (If any READE or READPE operations are used, no record blocking will occur for the input file. If any SETGT, SETLL, or CHAIN operations are used, no record blocking will occur unless the BLOCK(*YES) keyword is specified for the input file.)

When you specify BLOCK(*YES), record blocking occurs as described above except that the operations SETGT, SETLL, and CHAIN can be used with an input file and blocking will still occur (see condition 3c above).

To prevent the blocking of records, BLOCK(*NO) can be specified. No record blocking is done by the compiler.

COMMIT{(rpg_name)}

The COMMIT keyword allows the option of processing remote files under commitment control. An optional parameter, rpg_name, may be specified. The parameter is implicitly defined as a field of type indicator (that is, a character field of length one), and is initialized to '0'.

By specifying the optional parameter, the programmer can control whether commitment control is enabled at run time. If the parameter contains a '1', the file is opened with COMMIT on, otherwise the file is opened without COMMIT. The parameter must be set prior to opening the file. If the file is opened at program initialization, the parameter can be passed in through a parameter. If the file is opened explicitly, using the OPEN operation in the calculation specifications, it can be set prior to the OPEN operation.

Use the COMMIT and ROLBK operation codes to group changes to this file and other files currently under commitment control so that changes all happen together, or do not happen at all.

Note: If the file is already open with a shared open data path, the value for commitment control must match the value for the previous OPEN operation.

CVTHEX

The CVTHEX provides support for processing externally-described remote disk files containing database fields with CCSID 65535.

A CCSID value of 65535 implies that no conversion should be done when accessing the field data, but the traditional EBCDIC data often in these fields on the server isn't understood on the client workstation operating in ANSI.

When CVTHEX is specified for the file, any character fields with a CCSID of 65535 in the file will be converted to the workstation CCSID on input/output operations for use in the application. (The client-side conversion process uses the server connection job's CCSID in place of the field's 65535 CCSID to perform the conversion.)

Note: CVTHEX is not supported when compiling to JAVA.

DATFMT(format{separator})

The DATFMT keyword allows the specification of a default external date format and a default separator (which is optional) for *all* date fields in the program-described file. If the file, for which this keyword is specified, is indexed and the keyfield is a date, then this also provides the default format for the keyfield. The file can be either remote or local.

You can specify a different external format for individual input or output date fields in the file by specifying a date format/separator for the field on the corresponding input specification (positions 31–35) or output specification (position 53–57).

For date Input fields this specifies the default external date format/separator (Input specification positions 31-35).

For date Output fields this specifies the default external date format/separator (Output specification positions 53-57).

See “DATFMT(fmt{separator})” on page 227 for date formats and separators. For more information on external formats, see “Internal and External Formats” on page 103.

DEVMODE(name)

The DEVMODE keyword can be used for printer files to provide print settings.

Use the DEVMODE keyword to specify a data structure name containing a Windows operating system GDI DEVMODE structure to specify printer settings when the printer file is opened.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...

 * This example demonstrates using the DEVMODE keyword and a
 * DEVMODE data structure to set the printer to Landscape orientation

ffilo      o    f   80        printer         DEVMODE(pdevmode)
f                                             prtfmt(*sys)

 * The following initial settings select landscape orientation printing:
 * (For more information, see http://msdn.microsoft.com/ and search for
 *  DEVMODE.)

d pdevmode        ds           200

 * These next few fields need to be correctly initialized:
 *
d pdevname                1     32    inz(' ')
d pspecver               33     34i 0 inz(x'0401')
d pdrvver                35     36i 0 inz(0)
d pdmsize                37     38i 0 inz(148)
d pdrvextr               39     40i 0 inz(0)

 * This field contains the bit flags which indicate which of the
 * subsequent fields have valid settings to apply:
 * Bit Flag Settings:
 *  x'0001'  Orientation
 *  x'0100'  Number of Copies

 * Specify only the Orientation setting:   (x'00001' = 1)
d pfields                41     44i 0 inz(1)

 * Orientation:  1 = Portrait, 2 = Landscape
d porient                45     46i 0 inz(2)
 * Number of Copies:
d pcopies                55     56i 0
```

Figure 85. Setting Landscape Orientation for Printing

EOFMARK(*NONE)

Specify the EOFMARK(*NONE) keyword to omit the end-of-file marker from local disk files. The *NONE parameter is required.

EXTFILE(filename)

The EXTFILE keyword allows you to specify an actual filename to be opened at run time rather than supplying the name at compile time. The value can be a literal or a variable.

Notes:

1. If a variable name is used, it must be set before the file is opened. For files that are opened automatically at program initialization, the variable must be set in one of the following ways:
   - Using the INZ keyword on the D specification
   - Passing the value in as an entry parameter

Local Files

The file must be a local DISK or PRINTER file. The USROPN keyword must also be specified with the EXTFILE keyword.

Remote OS/400 Files

You can specify the value in any of the following forms:

```
filename
libname/filename
*LIBL/filename
```

Notes:

1. You cannot specify *CURLIB as the library name.
2. If you specify a file name without a library name, *LIBL is used.
3. The name must be in the correct case. For example, if you specify EXTFILE(filename) and variable filename has the value 'qtemp/myfile', the file will not be found. Instead, it should have the value 'QTEMP/MYFILE'.
4. This keyword is not used to find an externally-described file at compile time.

If you have specified an override for the file that RPG will open, that override will be in effect. In the following code, for the file named **INPUT** within the RPG program, the file that is opened at runtime depends on the value of the ***filename*** field.

```
Finput     if   e            disk    extfile(filename) remote
```

If the ***filename*** field has the value MYLIB/MYFILE at runtime, RPG will open the file MYLIB/MYFILE. If the command OVRDBF MYFILE OTHERLIB/OTHERFILE has been used, the actual file opened will be OTHERLIB/OTHERFILE. Note that any overrides for the name INPUT will be ignored, since INPUT is only the name used within the RPG source member.

EXTMBR(membername)

The EXTMBR keyword specifies which member of the file is opened. You can specify a member name, '*ALL', or '*FIRST'. Note that '*ALL' and '*FIRST' must be specified in quotes, since they are member "names", not RPG special words. The value can be a literal or a variable. The default is '*FIRST'.

The name must be in the correct case. For example, if you specify EXTMBR(mbrname) and variable mbrname has the value 'mbr1', the member will not be found. Instead, it should have the value 'MBR1'.

If a variable name is used, it must be set before the file is opened. For files that are opened automatically during program initialization, the variable must be set in one of the following ways:

- Using the INZ keyword on the D specification
- Passing the value in as an entry parameter

FORMLEN(number)

Use the FORMLEN keyword to specify the form length of a PRINTER file. The form length must be greater than or equal to 1 and less than or equal to 255. The parameter specifies the exact number of lines available on the form or page to be used. When the number of lines matches the FORMLEN, an automatic form feed is inserted.

IGNORE(recformat{:recformat...})

The IGNORE keyword lets you ignore a record format from an externally described file. The external name of the record format to be ignored is specified as the parameter recformat. One or more record formats can be specified, separated by colons (:). The program runs as if the specified record format(s) did not exist. All other record formats contained in the file will be included.

When the IGNORE keyword is specified for a file, the INCLUDE keyword cannot be specified.

INCLUDE(recformat{:recformat...})

The INCLUDE keyword specifies those record format names that are to be included. All other record formats contained in the file will be ignored. Multiple record formats can be specified, separated by colons (:).

When the INCLUDE keyword is specified for a file, the IGNORE keyword cannot be specified.

INFDS(DSname)

The INFDS keyword lets you define and name a data structure to contain the feedback information associated with the file. The data structure name is specified as the parameter for INFDS. If INFDS is specified for more than one file, each associated data structure must have a unique name. An INFDS can only be defined in the main source section.

INFSR(SUBRname)

The file exception/error subroutine specified as the parameter to this keyword may receive control following file exception/errors. The subroutine name may be *PSSR, which indicates the user defined program exception/error subroutine is to be given control for errors on this file.

The INFSR keyword cannot be specified if the file is to be accessed by a subprocedure

PLIST(Plist_name)

PLIST supplies, as its parameter, the name of the parameter list to be passed to the program for the SPECIAL file. The procedure is specified using the PROCNAME(proc_name) keyword. This entry is valid only when the device specified (positions 36 to 42) in the file-description line is SPECIAL. The parameters identified by this entry are added to the end of the parameter list passed by the program.

PREFIX(prefix{:nbr_of_char_replaced})

The PREFIX keyword is used to partially rename the fields in an externally-described file. The character string or character literal specified is prefixed to the names of all fields defined in all records of the file specified in positions 7–16. In addition, you can optionally specify a numeric value to indicate the number of characters, if any, in the existing name to be replaced. If the 'nbr_of_char_replaced' is not specified, then the string is attached to the beginning of the name.

If the 'nbr_of_char_replaced' is specified, it must be a numeric constant containing a value between 0 and 9 with no decimal places. For example, the specification PREFIX(YE:3) would change the field name 'YTDTOTAL' to 'YETOTAL'. Specifying a value of zero is the same as not specifying 'nbr_of_char_replaced' at all.

Rules:

- To explicitly rename a field on an Input specification when the PREFIX keyword has been specified for a file you must choose the correct field name to specify for the External Field Name (positions 21 - 30) of the Input specification. The name specified depends on whether the prefixed name has been used prior to the rename specification.
  - If there has been a prior reference made to the prefixed name, the prefixed name must be specified.
  - If there has not been a prior reference made to the prefixed name, the external name of the input field must be specified.

  Once the rename operation has been coded then the new name must be used to reference the input field. For more information, see External Field Name of the Input specification.
- The total length of the name after applying the prefix must not exceed the maximum length of a VisualAge RPG field name.
- The number of characters in the name to be prefixed must not be less than or equal to the value represented by the 'nbr_of_char_replaced' parameter. That is, after applying the prefix, the resulting name must not be the same as the prefix string.
- If the prefix is a character literal, it can end in a period. In this case, the field names must all be subfields of the same qualified data structure.
- If the prefix is a character literal, it must be uppercase.

Examples:

The following example uses prefix 'MYDS.' to associate the fields in MYFILE with the subfields of qualified data structure MYDS.

```
Fmyfile    if   e            disk    prefix('MYDS.') remote
D myds          e ds                  qualified extname(myfile)
```

The next example uses prefix 'MYDS.F2':3 to associate the fields in MYFILE with the subfields of qualified data structure MYDS. The subfields themselves are further prefixed by replacing the first three characters with 'F2'. The fields used by this file will be MYDS2.F2FLD1 and MYDS2.F2FLD2. (Data structure MYDS2 must be defined with a similar prefix. However, it is not exactly the same, since it does not include the data structure name.)

```
A          R REC
A            ACRFLD1       10A
A            ACRFLD2        5S 0
```

```
Fmyfile2   if   e            disk    prefix('MYDS2.F2':3) remote
D myds2         e ds                 qualified extname(myfile)
D                                    prefix('F2':3)
```

PROCNAME(proc_name)

When SPECIAL is the device entry (positions 36 through 42), the user-supplied code module specified as the parameter to PROCNAME handles the support for the special I/O device. See "Positions 36-42 (Device)" on page 242 and "PLIST(Plist_name)" on page 248 for more information.

PRTCTL(data_struct{:*COMPAT})

The PRTCTL keyword specifies the use of dynamic printer control. The data structure specified as the parameter data_struct refers to the forms control information and line count value. The PRTCTL keyword is valid only for a program described file.

The optional parameter *COMPAT indicates that the data structure layout is compatible with RPG III. If *COMPAT not specified, the extended length data structure must be used.

Extended Length PRTCTL Data Structure

A minimum of 15 bytes is required for this data structure. The layout of the PRTCTL data structure is as follows:

Data Structure Positions	Subfield Contents
1-3	A three-position character field that contains the space-before value (blank or 0-255)
4-6	A three-position character field that contains the space-after value (blank or 0-255)
7-9	A three-position character field that contains the skip-before value (valid entries: blank or 1-255)
10-12	A three-position character field that contains the skip-after value (blank or 1-255)
13-15	A three-digit numeric (zoned decimal) field with zero decimal positions that contains the current line count value.

*COMPAT PRTCTL Data Structure

Data Structure Positions	Subfield Contents
1	A one-position character field that contains the space-before value (blank or 0-3)
2	A one-position character field that contains the space-after value (valid entries: blank or 0-3)
3-4	A two-position character field that contains the skip-before value (blank, 1-99, A0-A9 for 100-109, B0-B2 for 110-112)
5-6	A two-position character field that contains the skip-after value (blank, 1-99, A0-A9 for 100-109, B0-B2 for 110-112)
7-9	A three-digit numeric (zoned decimal) field with zero decimal positions that contains the current line count value.

The values in the first four subfields of the extended length data structure are the same as those allowed in positions 40 through 51 (space and skip entries) of the output specifications. If the space and skip entries (positions 40 through 51) of the output specifications are blank, and if subfields 1 through 4 are also blank, the

default is to space 1 after. If the PRTCTL option is specified, it is used only for the output records that have blanks in positions 40 through 51. You can control the space and skip value (subfields 1 through 4) for the PRINTER file by changing the values in these subfields while the program is running.

Subfield 5 contains the current line count value. The VisualAge RPG compiler does not initialize subfield 5 until after the first output line is printed. The VisualAge RPG compiler then changes subfield 5 after each output operation to the file.

PRTFMT(*SYS | *TEXT)

The PRTFMT keyword with parameter *SYS can be used for printer files to specify the application should perform output to the printer through a device context and graphics device interface calls to the operating system, instead of the default raw text output.

After opening the printer file, the device context handle is copied to positions 81 to 84 of the printer file INFDS, for the application to reference in making it's own Windows GDI calls.

The default is *TEXT, where the application's text data is output directly.

Note: PRTFMT does not apply when compiling to JAVA.

RCDLEN(fieldname)

The RCDLEN keyword can be used for local DISK files. The field name parameter must be numeric with zero decimal places. For input files, the field name contains the length of the record that was read. For output files, the field name specifies the length of the record to be written. The record length specified in positions 23-27 defines the maximum field length. The RCDLEN must be less than or equal to this record length. The smallest record length that can be written to is zero. If the value specified with RECLEN is less than zero, it is rounded up to zero.

If the RCDLEN keyword is present, the file is treated as if it contains variable length records. If the keyword is not present, the file is treated as if it contains fixed length records.

Note: If the RCDLEN field is set on output, it overrides the length of any data structure being used.

RECNO(fieldname)

This keyword is optional for DISK files to be processed by relative-record number. The RECNO keyword must be specified for output files processed by relative-record number, output files that are referenced by a random WRITE calculation operation, or output files that are used with ADD on the output specifications.

Note: If you do not specify the RECNO keyword, records blocking occurs.

The RECNO keyword can be specified for input/update files. The relative-record number of the record retrieved is placed in the 'fieldname', for all operations that reposition the file (such as READ, SETLL, or OPEN). It must be defined as numeric with zero decimal positions. The field length must be sufficient to contain the longest record number for the file.

When the RECNO keyword is specified for input or update files with file-addition ('A' in position 20), the value of the fieldname parameter must refer to a relative-record number of a deleted record, for the output operation to be successful.

Note: The RECNO keyword is ignored if you are writing (WRITE) to a local file.

REMOTE

The REMOTE keyword specifies that the disk device resides on an iSeries server.

RENAME(Ext_format:Int_format)

The RENAME keyword allows you to rename record formats in an externally described file. The external name of the record format that is to be renamed is entered as the Ext_format parameter. The Int_format parameter is the name of the record as it is used in the program. The external name is replaced by this name in the program.

To rename all fields by adding a prefix, use the PREFIX keyword.

TIMFMT(format{separator})

The TIMFMT keyword allows the specification of a default external time format and a default separator (which is optional) for *all* time fields in the program described fields. If the file, on which this keyword is specified, is indexed and the keyfield is a time, then the time format specified also provides the default format for the keyfield. The file can either be local or remote.

You can specify a different external format for individual input or output time fields in the file by specifying a time format/separator for the field on the corresponding input specification (positions 31-35)or output specification (positions 53–57).

See Table 18 on page 136 for valid format and separators. For more information on external formats see "Internal and External Formats" on page 103

USROPN

The USROPN keyword causes the file not to be opened at program initialization. This gives the programmer control of the file's first open. The file must be explicitly opened using the OPEN operation in the calculation specifications. This keyword is not valid for input files designated as table files.

The USROPN keyword is required for programmer control of the first file opening. For example, if a file is opened and later closed by the CLOSE operation, the file can be reopened (using the OPEN operation) without having specified the USROPN keyword on the file description specification.

File Types and Processing Methods

The following table shows the valid entries for positions 28, 34, and 35 of the file-description specifications for the various file types and processing methods. The methods of disk file processing include:

- Relative-record-number processing
- Consecutive processing
- Sequential-by-key processing
- Random-by-key processing

Note: Local DISK files can only be processed sequentially or by relative record.

Table 28. Processing Methods for DISK Files

Access	Method	Opcode	Position 28	Position 34	Position 35	Explanation
Random	RRN	CHAIN	Blank	Blank	Blank	Access by physical order of records
Sequential	Key	READ READE READP READPE	Blank	K	Blank	Access by key sequentially
Sequential	RRN	READ	Blank	Blank	Blank	Access sequentially
Random	Key	CHAIN	Blank	K	Blank	Access by key randomly

Chapter 18. Definition Specifications

Definition Specifications can be used to define data structures, data-structure subfields, prototypes, procedure interfaces, prototyped parameters, standalone fields, named constants, and message windows.

Depending on where the definition occurs, there are differences both in what can be defined and also the scope of the definition. Specify the type of definition in positions 24 through 25, as follows:

Entry	Definition Type
Blank	A data structure subfield or parameter definition
C	Named constant
DS	Data structure
PI	Procedure interface
PR	Prototype
S	Standalone field

Definitions of data structures, prototypes, and procedure interfaces end with the first definition specification with non-blanks in positions 24-25, or with the first specification that is not a definition specification.

Definition specifications can appear in two places within a module or program: in the main source section and in a subprocedure. Within the main source section, you define all global definitions. Within a subprocedure, you define the procedure interface and its parameters as required by the prototype. You also define any local data items that are needed by the prototyped procedure when it is processed. Any definitions within a prototyped procedure are local. They are not known to any other procedures (including the main procedure). For more information on the structure of the main source section and how the placement of definitions affects scope, see "Placement of Definitions and Scope" on page 256.

On the definition specification, arrays and tables can be defined as either a data-structure subfield or a standalone field. For additional information on defining and using arrays and tables, see Chapter 12, "Using Arrays and Tables," on page 171.

Built-in functions (BIF) can be specified on definition specifications in the keyword field as a parameter to a keyword. A built-in function is allowed on the definition specification only if the values of all arguments are known at compile-time. All arguments for a BIF must be defined earlier in the specifications when specified as parameters for the definition specification keywords DIM, OCCURS, OVERLAY, and PERRCD. For further information on using built-in functions, see "Built-In Functions (Alphabetically)" on page 405.

For further information on data structures, constants, data types, and data formats, see Chapter 9, "Data Types and Data Formats," on page 103, Chapter 11, "Data Structures," on page 157, and Chapter 10, "Literals and Named Constants," on page 149. For more information on prototypes, see "Prototypes and Parameters" on page 71.

Placement of Definitions and Scope

Depending on where a definition occurs, it will have different scope. **Scope** refers to the range of source lines where a name is known. There are two types of scope: global and local. Figure 86 shows how the placement of definitions in a module is related to scope. Figure 87 on page 257 shows the layout of the main source section for each possible compilation target: component, NOMAIN DLL, or EXE.

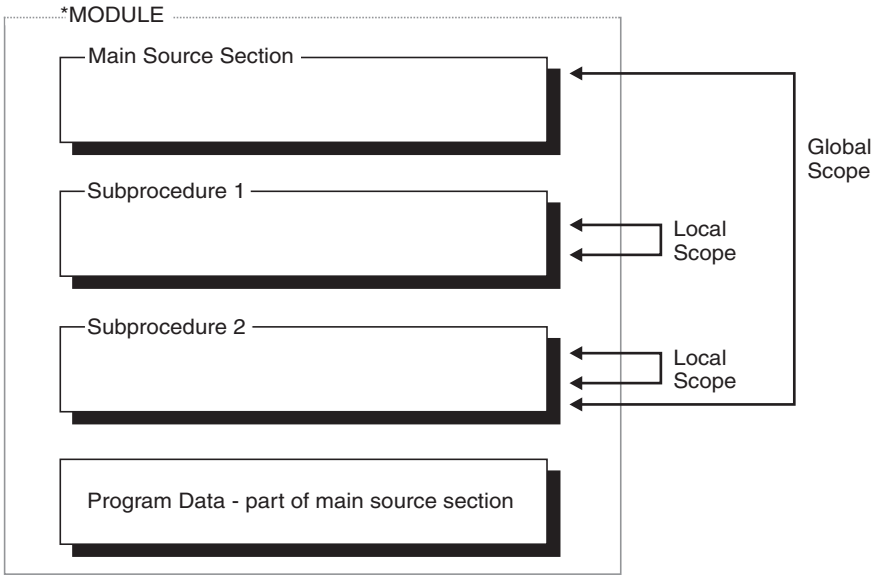


Figure 86. Scope of Definitions

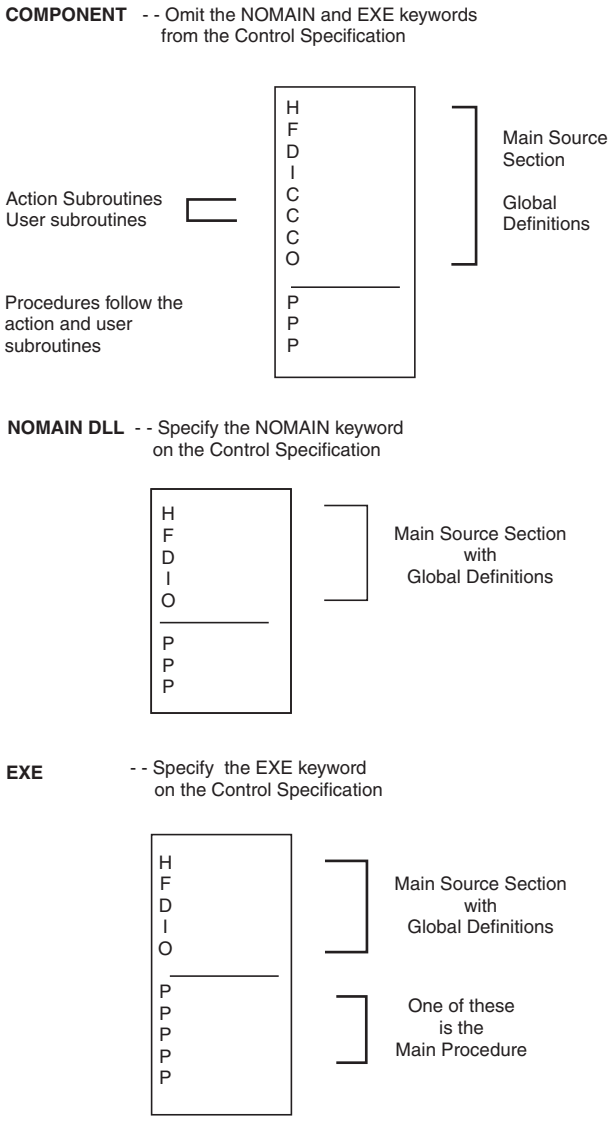


Figure 87. Main Source Section for Each Compilation Target

In general, all items that are defined in the main source section are global, and therefore, known throughout the module. **Global definitions** are definitions that can be used by both the statements in the main procedure and any subprocedures within the module.

Items in a subprocedure, on the other hand, are local. **Local definitions** are definitions that are known only inside that subprocedure. If an item is defined with the same name as a global item, then any references to that name inside the subprocedure will use the local definition.

However, note the following exceptions:

- Subroutine names and tag names are known only to the procedure in which they are defined. This includes subroutine or tag names that defined in the main procedure.
- All fields specified on input and output specifications are global. For example, if a subprocedure does an operation using a record format, say a WRITE

operation, the global fields will be used even if there are local definitions with the same names as the record format fields. This rule also applies to the READ and WRITE of windows.

Sometimes you may have a mix of global and local definitions. For example, KLISTs and PLISTs can be global or local. The fields associated with *global* KLISTs and PLISTs contain only global fields. The fields associated with *local* KLISTs and PLISTs can contain both global and local fields. For more information on the behavior of KLISTs and KFLDs inside subprocedures, see "Scope of Definitions" on page 66.

Storage of Definitions

Local definitions use automatic storage. **Automatic storage** is storage that exists only for the duration of the call to the procedure. Variables in automatic storage do not save their values across calls.

Global definitions, on the other hand, use static storage. **Static storage** is storage that has a constant location in memory for all calls of a program or procedure. It keeps its value across calls.

Specify the STATIC keyword to indicate that a local field definition use static storage, in which case it will keep its value on each call to the procedure. If the keyword STATIC is specified, the item will be initialized at module initialization time.

Using automatic storage reduces the amount of storage that is required at run time by the program. The storage is reduced largely because automatic storage is only allocated while the procedure is running. On the other hand, all static storage associated with the program is allocated when the program starts, even if no procedure using the static storage is ever called.

Definition Specification Statement

The general layout for the definition specification is as follows:

- The definition specification type (D) is entered in position 6
- The non-commentary part of the specification extends from position 7 to position 80
  - The fixed-format entries extend from positions 7 to 42
  - The keyword entries extend from positions 44 to 80
- The comments section of the specification extends from position 81 to position 100

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 88. Definition Specification Layout

Definition-Specification Keyword Continuation Line

If additional space is required for keywords, the keywords field can be continued on subsequent lines as follows:

- Position 6 of the continuation line must contain a D
- Positions 7 to 43 of the continuation line must be blank
- The specification continues on or past position 44

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
D.....................................Keywords++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 89. Definition-Specification Keyword Continuation Line Layout

Definition Specification Continued Name Line

A name that is up to 15 characters long can be specified in the Name entry of the definition specification without requiring continuation. Any name (even one with 15 characters or fewer) can be continued on multiple lines by coding an ellipsis (...) at the end of the partial name. A name definition consists of the following parts:

1. Zero or more continued name lines. Continued name lines are identified as having an ellipsis as the last non-blank character in the entry. The name must begin within positions 7 to 21 and may end anywhere up to position 77 (with an ellipsis ending in position 80). There cannot be blanks between the start of the name and the ellipsis character. If any of these conditions is not true, the line is parsed as a main definition line.
2. One main definition line, containing a name, definition attributes, and keywords. If a continued name line is coded, the Name entry of the main definition line may be left blank.
3. Zero or more keyword continuation lines.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
DContinuedName+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 90. Definition Specification Continued Name Line Layout

Position 6 (Form Type)

A D must be entered in this position for definition specifications.

Positions 7-21 (Name)

Entry	Explanation
Name	The name of the data structure, data-structure subfield, standalone field, named constant, local program, and parameters for the local program to be defined.
Blank	Specifies filler fields in data-structure subfield definitions, or an unnamed data structure in data-structure definitions.

Use positions 7-21 to specify the name of the data item being defined. The name can begin in any position in the space provided. Indenting can be used to indicate the shape of data in data structures.

For continued name lines, a name is specified in positions 7 through 80 of the continued name lines and positions 7 through 21 of the main definition line. As with the traditional definition of names, case of the characters is not significant.

For an externally-described subfield, a name specified here replaces the external-subfield name specified on the EXTFLD keyword.

For a prototype parameter definition, the name entry is optional. If a name is specified, the name is ignored. (A prototype parameter is a definition specification with blanks in positions 24-25 that follows a PR specification or another prototype parameter definition.)

If you are defining a prototype and the name specified in positions 7-21 cannot serve as the external name of the procedure, use the EXTPROC keyword to specify the valid external name. For example, the external name may be required to be in lower case, because you are defining a prototype for a procedure written in C.

Position 22 (External Description)

This position identifies a data structure or data-structure subfield as externally described. If a data structure or subfield is not defined on this specification, then this field must be left blank.

Entry	Explanation for Data Structures
E	Identifies a data structure as externally described: subfield definitions are defined externally. If the EXTNAME keyword is not specified, positions 7-21 must contain the name of the externally described file containing the data structure definition.
Blank	Program described: subfield definitions for this data structure follow this specification.

Entry	Explanation for Subfields
E	Identifies a data-structure subfield as externally described. The specification of an externally-described subfield is necessary only when keywords such as EXTFLD and INZ are required.
Blank	Program described: the data-structure subfield is defined on this specification line.

Position 23 (Type of Data Structure)

This entry is used to identify the type of data structure being defined. If a data structure is not being defined, this entry must be left blank.

Entry	Explanation
Blank	The data structure being defined is not a program status or data-area data structure; or a data structure is not being defined on this specification.
S	Program status data structure. Only one data structure may be designated as the program status data structure.
U	Data-area data structure. The data area is retrieved at initialization and is rewritten at the end of the program: • If the DTAARA keyword is specified, the parameter to the DTAARA keyword is used as the name of the external data area. If the name is a variable, the value must be set before the program begins. This can be done by: – Passing the variable as a parameter. – Explicitly initializing the variable with the INZ keyword. – Sharing the variable with another module using the IMPORT and EXPORT keywords, and ensuring the value is set prior to the call. • If the DTAARA keyword is not specified, the name in positions 7-21 is used as the name of the external data area.

Positions 24-25 (Type of Definition)

Entry	Explanation
Blank	The specification defines a data structure subfield or a parameter within a prototype or procedure interface definition.
C (in column 24)	The specification defines a constant. Position 25 must be blank.
DS	The specification defines a data structure.
M (in column 24)	The specification defines a message window for use with the DSPLY operation code. Position 25 must be blank.
PI	The specification defines a procedure interface, and the return value if any.
PR	The specification defines a prototype for a call to a local EXE, CMD, or BAT file. The PR specification is followed by zero or more parameter definitions (a blank in positions 24-25), indicating the number and type of parameters required by the program. A prototype definition ends with the first definition specification with non-blanks in positions 24-25, or with the first specification that is not a definition specification.
S (in column 24)	The specification defines a standalone field, array or table. Standalone fields allow you to define individual work fields, without requiring the definition of a data structure. The following is allowed for standalone fields: • A standalone field has a specifiable internal data type. • A standalone field may be defined as an array, table or field. • Only length notation is allowed.

Definitions of data structures, prototypes, and procedure interfaces end with the first definition specification with non-blanks in positions 24-25, or with the first specification that is not a definition specification.

Named constant and standalone-field definition specifications may not be included within definition specifications for a data structure and its subfields.

For a list of valid keywords, grouped according to type of definition, see "Summary According to Definition Specification Type" on page 295.

Positions 26-32 (From Position)

Positions 26-32 may only contain an entry if the location of a subfield within a data structure is being defined.

Entry **Explanation**

Blank A blank FROM position indicates that the value in the TO/LENGTH field specifies the length of the subfield, or that a subfield is not being defined on this specification line.

nnnnnnn
Absolute starting position of the subfield within a data structure. The value specified must be from 1 to 65535 for a named data structure (and from 1 to 9999999 for an unnamed data structure), and right-justified in these positions.

Reserved Words
Reserved words for the program status data structure or for a file information data structure are allowed (left-justified) in the FROM-TO/LENGTH fields (positions 26-39). These special reserved words define the location of the subfields in the data structures. Reserved words for the program status data structure are *STATUS, *PROC, *PARM, and *ROUTINE. Reserved words for the file information data structure (INFDS) are *FILE, *RECORD, *OPCODE, *STATUS, and *ROUTINE.

Positions 33-39 (To Position/Length)

Entry **Explanation**

Blank If positions 33-39 are blank:
- A named constant is being defined on this specification line, or
- The standalone field or subfield is being defined LIKE another field, or
- The standalone field or subfield is of a type where a length is implied, or,
- The subfield's attributes are defined elsewhere, or
- A data structure is being defined. The length of the data structure is the maximum value of the subfield To-Positions. The data structure may be defined using the LIKEDS or LIKEREC keyword.

nnnnnnn
Positions 33-39 may contain a (right-justified) numeric value, from 1 to 65535 for a named data structure (and from 1 to 9999999 for an unnamed data structure), as follows:
- If the From field (position 26-32) contains a numeric value, then a numeric value in this field specifies the absolute end position of the subfield within a data structure.
- If the From field is blank, a numeric value in this field specifies :
  - The length of the entire data structure, or
  - The length of the standalone field, or
  - the length of the parameter, or
  - The length of the subfield.

    Within the data structure, this subfield is positioned such that its starting position is greater than the maximum to-position of all previously defined subfields in the data structure. Padding is inserted if the subfield is defined with type basing pointer or procedure pointer to ensure that the subfield is aligned properly.

Note: For graphic or UCS-2 fields, the number specified here is the number of graphic or UCS-2 characters, NOT the number of bytes (1 graphic or UCS-2 character = 2 bytes). For numeric fields, the number specified here is the number of digits (for packed and zoned numeric fields: 1-31; for binary numeric fields: 1-9; for integer and unsigned numeric fields: 3, 5, 10, or 20).

+|-nnnnn

This entry is valid for standalone fields or subfields defined using the LIKE keyword. The length of the standalone field or subfield being defined on this specification line is determined by adding or subtracting the value entered in these positions to the length of the field specified as the parameter to the LIKE keyword.

Note: For graphic or UCS-2 fields, the number specified here is the number of graphic or UCS-2 characters, NOT the number of bytes (1 graphic or UCS-2 character = 2 bytes). For numeric fields, the number specified here is the number of digits.

Reserved Words

If positions 26-32 are used to enter special reserved words, this field becomes an extension of the previous one, creating one large field (positions 26-39). This allows for reserved words, with names longer than 7 characters in length, to extend into this field. See "Positions 26-32 (From Position)" on page 262.

Position 40 (Internal Data Type)

This entry allows you to specify how a standalone field or data-structure subfield is stored internally. This entry pertains strictly to the internal representation of the data item being defined, regardless of how the data item is stored externally (that is, if it is stored externally). To define variable-length character, graphic, and UCS-2 formats, you must specify the keyword VARYING; otherwise, the format will be fixed length.

Entry	Explanation
Blank	If the LIKE keyword is not specified: the item is being defined as character if the decimal positions entry is blank. If the decimal positions entry is not blank, the item is defined as packed numeric if it is a standalone field, or as zoned numeric if it is a subfield. **Note:** The entry must be blank whenever the LIKE, LIKEDS and LIKEREC keywords are specified.
A	Character (Fixed or Variable-length format)
N	Character (Indicator format)
C	UCS-2 (Fixed or Variable-length format)
G	Graphic (Fixed or Variable-length format)
T	Time
D	Date
Z	Timestamp
O	Object
P	Numeric (Packed decimal format)

B	Numeric (Binary format)
I	Numeric (Integer format)
S	Numeric (Zoned format)
U	Numeric (Unsigned format)
F	Numeric (Float format)
O	Object (for Java™ applications only)
*	Basing pointer or procedure pointer

Positions 41-42 (Decimal Positions)

Positions 41-42 are used to indicate the number of decimal positions in a numeric subfield or standalone field. If the field is numeric, there must always be an entry in these positions; if there are no decimal positions, enter a 0.

Entry	Explanation
Blank	The value is not numeric or has been defined with the LIKE keyword.
0-31	Decimal positions: the number of positions to the right of the decimal in a numeric field.

This entry can only be supplied in combination with the TO/Length field. If the TO/Length field is blank, the value of this entry is defined somewhere else in the program (for example, through an externally described database file).

Position 43 (Reserved)

Position 43 must be blank.

Positions 44-80 (Keywords)

Positions 44 to 80 are provided for definition-specification keywords. Keywords are used to describe and define data and its attributes. See "Definition-Specification Keywords" for a description of each keyword.

Use this area to specify any keywords necessary to fully define the field.

Definition-Specification Keywords

Definition-specification keywords can have no parameters, optional parameters, or required parameters. The syntax for keywords is as follows:

```
Keyword(parameter1 : parameter2)
```

where:
- Parameter(s) are enclosed in parentheses ( ).

  Note: Do not specify parentheses if there are no parameters.
- Colons (:) are used to separate multiple parameters.

The following notational conventions are used to show which parameters are optional and which are required:
- Braces { } indicate optional parameters or optional elements of parameters.
- An ellipsis (...) indicates that the parameter can be repeated.
- A colon (:) separates parameters and indicates that more than one may be specified. All parameters separated by a colon are required unless they are enclosed in braces.

- A vertical bar (|) indicates that only one parameter may be specified for the keyword.
- A blank separating keyword parameters indicates that one or more of the parameters may be specified.

Note: Braces, ellipses, and vertical bars are not a part of the keyword syntax and should not be entered into your source.

If additional space is required for keywords, the keyword field can be continued on subsequent lines. See "Definition-Specification Keyword Continuation Line" on page 259 and "Definition Specification Keyword Field" on page 218.

ALIGN

The ALIGN keyword is used to align float, integer, and unsigned subfields. When ALIGN is specified, 2-byte subfields are aligned on a 2-byte boundary, 4-byte subfields are aligned on a 4-byte boundary and 8-byte subfields are aligned on an 8-byte boundary. Alignment may be desired to improve performance when accessing float, integer, or unsigned subfields.

Specify ALIGN on the data structure definition. However, you cannot specify ALIGN for either the file information data structure (INFDS) or the program status data structure (PSDS).

Alignment occurs only to data structure subfields defined with length notation and without the keyword OVERLAY. A diagnostic message is issued if subfields that are defined either with absolute notation or using the OVERLAY keyword are not properly aligned.

Pointer subfields are always aligned on a 4-byte boundary whether or not ALIGN is specified.

See "Aligning Data Structure Subfields" on page 160 for more information.

ALT(array_name)

The ALT keyword indicates that the compile-time array, pre-runtime array, or table is in alternating format.

The array defined with the ALT keyword is the alternating array and the array name specified as the parameter is the main array. The alternate array definition may precede or follow the main array definition.

The keywords on the main array define the loading for both arrays. The initialization data is in alternating order, beginning with the main array, as follows: main/alt/main/alt/...

In the alternate array definition, the PERRCD, FROMFILE, TOFILE, and CTDATA keywords are not valid.

ASCEND

The ASCEND keyword describes the sequence of the data in an array or table loaded at pre-runtime or compile time. See "DESCEND" on page 269.

Ascending sequence means that the array or table entries must start with the lowest data entry (according to the default ASCII collating) and go to the highest. Items with equal value are allowed.

A pre-runtime array or table is checked for the specified sequence at the time the array or table is loaded with data. If the array or table is out of sequence, control passes to the exception/error handling routine. A run-time array (loaded by input and/or calculation specifications) is not sequence checked.

A sequence (ascending or descending) must be specified if the LOOKUP operation, %LOOKUPxx built-in, or %TLOOKUPxx built-in is used to search an array or table for an entry to determine whether the entry is high or low compared to the search argument.

If the SORTA operation code is used with an array, and no sequence is specified, an ascending sequence is assumed.

BASED(basing_pointer_name)

When the BASED keyword is specified for a data structure or standalone field, a basing pointer is created using the name specified as the keyword parameter. This basing pointer holds the address (storage location) of the based data structure or standalone field being defined. In other words, the name specified in positions 7-21 is used to refer to the data stored at the location contained in the basing pointer.

Note: Before the based data structure or standalone field can be used, the basing pointer must be assigned a valid address.

If an array is defined as a based standalone field it must be a run-time array.

If a based field is defined within a subprocedure, then both the field and the basing pointer are local.

BUTTON(button1:button2....)

The BUTTON keyword defines the buttons on the message window that are specified in Factor 2 of the DSPLY operation code. You can specify a maximum of 3 button parameters per keyword. The valid button combinations are:

*OK	*OK: *CANCEL	*RETRY: *CANCEL
*YESBUTTON: *NOBUTTON	*RETRY: *ABORT: *IGNORE	*YESBUTTON: *NOBUTTON: *CANCEL

This keyword cannnot be used if the MSGDATA, MSGNBR, or MSGTEXT keywords are used.

CCSID(number | *DFT)

This keyword sets the CCSID for graphic and UCS-2 definitions.

number must be an integer between 0 and 65535. It must be a valid graphic or UCS-2 CCSID value. Valid UCS-2 CCSIDs are 13488 and 17584.

For program-described fields, CCSID(number) overrides the defaults set on the control specification with the CCSID(*GRAPH: *SRC), CCSID(*GRAPH: number), or CCSID(*UCS2: number) keyword.

CCSID(*DFT) indicates that the default CCSID for the module is to be used. This is useful when the LIKE keyword is used since the new field would otherwise inherit the CCSID of the source field.

If the keyword is not specified, the default graphic or UCS-2 CCSID of the module is assumed. (This keyword is not allowed for graphic fields when CCSID(*GRAPH : *IGNORE) is specified or assumed).

If this keyword is not specified and the LIKE keyword is specified, the new field will have the same CCSID as the LIKE field.

CLASS(*JAVA:class_name)

This keyword indicates the class for an object definition.

To declare fields that can store objects, specify O in column 40 of the D-specification and use the CLASS keyword to provide the class of the object. The CLASS keyword requires two parameters:

```
CLASS(*JAVA:class_name)
```

*JAVA identifies the object as a Java object. class_name specifies the class of the object. The class name must be a character literal and fully qualify the Java class. The class name is case sensitive.

Fields of type O cannot be defined as subfields of data structures. It is possible to have arrays of type O fields, but tables of type O are not allowed because tables have to be preloaded at run time.

The following keywords cannot be used with the CLASS keyword:

```
ALIGN, ALT, ASCEND, BASED, BUTTON, CLTPGM, CONST, CTDATA, DATFMT,
DESCEND, DTAARA, EXTFLD, EXTFMT, EXTNAME, FROMFILE, INZ, LINKAGE,
MSGDATA, MSGNBR, MSGTEXT, MSGTITLE, NOOPT, NOWAIT, OCCURS, OPTIONS,
OVERLAY, PACKEVEN, PERRCD, PREFIX, PROCPTR, STYLE, TIMFMT, TOFILE,
VALUE, VARYING
```

For more information on calling Java methods and examples, see the *Programming with VisualAge RPG* manual.

CLTPGM(program name)

The CLTPGM keyword is used to specify the name of the local program called by the VisualAge RPG program, using the CALLP operation.

The local program that is called can be an EXE, a PIF, a COM, or a BAT file.

The default extension is EXE.

Note: A definition specification must be coded for each parameter.

CONST(constant)

The CONST keyword is used to specify the value of a named constant. This keyword is optional (the constant value can be specified with or without the CONST keyword), and is only valid for named constant definitions (C in position 24).

The parameter must be a literal, figurative constant, or built-in-function. The constant may be continued on subsequent lines by adhering to the appropriate continuation rules. See “Continuation Rules” on page 215.

If a named constant is used as a parameter for the keywords DIM, OCCURS, PERRCD, or OVERLAY, the named constant must be defined prior to its use.

When specifying a read-only reference parameter, you specify the keyword CONST on the definition specification of the parameter definition on both the prototype and procedure interface. No parameter to the keyword is allowed.

When the keyword CONST is specified, the compiler may copy the parameter to a temporary and pass the address of the temporary. Some conditions that would cause this are: the passed parameter is an expression or the passed parameter has a different format.

Attention!

Do not use this keyword on a prototype definition unless you are sure that the parameter will not be changed by the called program or procedure.

If the called program or procedure is compiled using a procedure interface with the same prototype, you do not have to worry about this, since the compiler will check this for you.

Although a CONST parameter cannot be changed by statements within the procedure, the value may be changed as a result of statements outside of the procedure, or by directly referencing a global variable.

Passing a parameter by constant value has the same advantages as passing by value. In particular, it allows you to pass literals and expressions.

CTDATA

The CTDATA keyword indicates that the array or table is loaded using compile-time data. The data is specified at the end of the program following the ** or **CTDATA(array/table name) specification.

When an array or table is loaded at compilation time, it is compiled along with the source program and included in the program. Such an array or table does not need to be loaded separately every time the program is run.

DATFMT(format{separator})

The DATFMT keyword specifies the internal date format for a Date field and optionally the separator character. This keyword is automatically generated for an externally described data structure subfield of type Date and determined at compile time.

This keyword can be used when defining CALLP parameters.

See "DATFMT(fmt{separator})" on page 227.

The hierarchy used when determining the internal format and separator for a date array or field is:
1. From the DATFMT keyword specified on the definition specification
2. From the DATFMT keyword specified in the control specification
3. *ISO

DESCEND

The DESCEND keyword describes the sequence of the data in an array or table loaded at pre-runtime or compile time. See "ASCEND" on page 265.

Descending sequence means that the array or table entries must start with the highest data entry (according to the collating sequence) and go to the lowest. Items with equal value are allowed.

A pre-runtime array or table is checked for the specified sequence at the time the array or table is loaded with data. If the array or table is out of sequence, control passes to the exception/error handling routine. A run-time array (loaded by input and/or calculation specifications) is not sequence checked.

A sequence (ascending or descending) must be specified if the LOOKUP operation, %LOOKUPxx built-in, or %TLOOKUPxx built-in is used to search an array or table for an entry to determine whether the entry is high or low compared to the search argument.

If the SORTA operation code is used with an array, and no sequence is specified, an ascending sequence is assumed.

DIM(numeric_constant)

The DIM keyword defines the number of elements in an array, table, a prototyped parameter, array data structure, or a return value on a prototype or procedure-interface definition.

The numeric constant must have zero (0) decimal positions. It can be a literal, a named constant or a built-in function.

The constant value does not need to be known at the time the keyword is processed, but the value must be known at compile-time.

This keyword can be used when defining CALLP parameters.

When DIM is specified on a data structure definition, the data structure must be a qualified data structure, and subfields must be referenced as fully qualified names, i.e. "dsname(x).subf". Other array keywords, such as CTDATA, FROMFILE, TOFILE, and PERRCD are not allowed with an array data structure definition.

DLL(name)

The DLL keyword, together with the LINKAGE keyword, is used to prototype a procedure that calls functions in Windows® DLLs, including Windows APIs.

The following example shows how to code the prototype and call to the Windows API GetCurrentDirectory:

```
D GetCurDir       PR            10I 0 ExtProc('GetCurrentDirectoryA')

D                                     DLL('KERNEL32.DLL')
D                                     Linkage(*StdCall)
D                                10I 0 Value
D                               255A
D CurDir          S             255A
D CurDirSiz       S              10I 0 Inz(%Size(CurDir))
D RCLong          S              10I 0

C                   Eval      RCLong = GetCurDir(CurDirSiz:CurDir)
```

The *A* in the external procedure name (GetCurrentDirectory**A**) indicates that the single-byte version of the DLL is being called. To call the unicode version, specify a *W*.

DTAARA{(*VAR:)data_area_name}

The DTAARA keyword is used to associate a standalone field, data structure, data-structure subfield, or data-area data structure with an external data area. The DTAARA keyword has the same function as the *DTAARA DEFINE operation code. See "Defining a Field as a Data Area" on page 548.

If data_area_name is not specified then the name specified in positions 7-21 is also the data area name. If data_area_name is specified, then it must be a data area name.

If data_area_name is not specified, then the name specified in positions 7-21 is also the name of the external data area.

If the parameter is not specified, then the data-structure name must be.

If *VAR is not specified, the data_area_name parameter can be either a name or a literal. If a name is specified, the name of the parameter of DTAARA is used as the name of the data area. For example, DTAARA(MYDTA) means that the data area *LIBL/MYDTA will be used at runtime. It must be a valid data area name, including *LDA (for the local data area) and *PDA (for the program initialization

parameters data area). If a literal is specified, the value of the literal is used as the name of the data area. For example, DTAARA('LIB/DTA') will use data area DTA in library LIB, at runtime.

If *VAR is specified, the value of data_area_name is used as the data area name. This value can be:
- A named constant whose value is the name of the data area.
- A character variable that will hold the name of the data area at runtime.

You can specify the value in any of the following forms:

```
dtaaraname
libname/dtaaraname
*LIBL/dtaaraname
```

Notes:

1. You cannot specify *CURLIB as the library name.
2. If you specify a data area name without a library name, *LIBL is used.
3. The name must be in the correct case. For example, if you specify DTAARA(*VAR:dtaname) and variable dtaname has the value 'qtemp/mydta', the data area will not be found. Instead, it should have the value 'QTEMP/MYDTA'.

Attention!
If DTAARA(*VAR) keyword is used with a UDS data area, and the name is a variable, then this variable must have the value set before the program starts. This can be done by initializing the variable, or passing the variable as an entry parameter.

When the DTAARA keyword is specified, the IN, OUT, and UNLOCK operation codes can be used on the data area.

EXTFLD(field_name)

The EXTFLD keyword is used to rename a subfield in an externally described data structure. The field_name parameter is the external name of the subfield. The name of the program to be used is specified in the Name field (positions 7-21).

The keyword is optional. If not specified, the name extracted from the external definition is used as the data-structure subfield name.

If the PREFIX keyword is specified for the data structure, the prefix will not be applied to fields renamed with EXTFLD.

EXTFMT(code)

The EXTFMT keyword specifies the external data type for compile-time and pre-runtime numeric arrays and tables. The external data type is the format of the data in the records in the file. This entry has no effect on the format used for internal processing (internal data type) of the array or table in the program.

The possible values for the parameter are:

S The data for the array or table is in zoned decimal format.

P The data for the array or table is in packed decimal format.

B The data for the array or table is in binary format.

C The data for the array or table is in UCS-2 format.

I The data for the array or table is in integer format.

L The data for a numeric array or table element has a preceding (left) plus or minus sign.

R The data for a numeric array or table element has a following (right) plus or minus sign.

U The data for the array or table is in unsigned format.

F The data for the array or table is in float numeric format.

Notes:

1. If the EXTFMT keyword is not specified, the external format defaults to 'S' for non-float arrays and tables, and to the external display float representation for float pre-runtime arrays and tables.
2. For compile-time arrays and tables, the only values allowed are S, L, and R, unless the data type is float, in which case the EXTFMT keyword is not allowed.
3. EXTFMT(I) or EXTFMT(U) is not allowed for arrays defined with more than 10 digits. Arrays defined as having 1 to 5 digits will occupy 2 bytes. Arrays defined as having 6 to 10 digits will occupy 4 bytes.
4. When EXTFMT(I) or EXTFMT(U) is used, arrays defined as having 1 to 5 digits will occupy 2 bytes per element. Arrays defined as having 6 to 10 digits will occupy 4 bytes per element. Arrays defined as having 11 to 20 digits will occupy 8 bytes per element.
5. The default external format for UCS-2 arrays is character. The number of characters allowed for UCS-2 compile-time data is the number of double-byte characters in the UCS-2 array.
6. The EXTFMT keyword cannot be used if the data for the array or table resides on the workstation.

EXTNAME(file-name{:format-name}{:*ALL|*INPUT|*OUTPUT|*KEY})

The EXTNAME keyword specifies the name of the file which contains the field descriptions used as the subfield description for the data structure being defined.

The file_name parameter is required. Optionally, a format name may be specified to direct the compiler to a specific format within a file. If format_name parameter is not specified, the first record format is used.

The last parameter specifies which fields in the external record to extract:

- ***ALL** extracts all fields.
- ***INPUT** extracts just input capable fields.
- ***OUTPUT** extracts just output capable fields.
- ***KEY** extracts just key fields.

If this parameter has not specified, the compiler extracts the fields of the input buffer.

Notes:

1. If the format-name is not specified,, the record defaults to the first record in the file.
2. For *INPUT and *OUTPUT, subfields included in the data structure occupy the same start positions as in the external record description.

If the data structure definition contains an E in column 22, and the EXTNAME keyword is not specified, the name specified in positions 7-21 is used.

The compiler generates the following Definition specification entries for all fields of the externally described data structure:
- Subfield name: this name is the same as the external name, unless renamed by keyword EXTFLD or the PREFIX keyword is used to apply a prefix
- Subfield length
- Subfield internal data type: this name is the same as the External type, unless the CVTOPT control specification keyword is specified for the type. In that case the data type will be character.

All data structure keywords except LIKEDS and LIKEREC are allowed with the EXTNAME keyword.

EXTPGM(name)

The EXTPGM keyword indicates the external name of the remote program on an iSeries server whose prototype is being defined. The name can be a character constant or a character variable.

If neither EXTPGM or EXTPROC is specified, then the compiler assumes that you are defining a prototype for a procedure, and assigns it the external name found in positions 7-21.

Any parameters defined by a prototype with EXTPGM must be passed by reference. In addition, you cannot define a return value.

EXTPROC({*JAVA:class-name:}name)

The EXTPROC keyword can have one of the following formats:

EXTPROC(*JAVA:class-name:name)
Specifies a method that is written in Java, or an RPG native method to be called by Java. The first parameter is *JAVA. The second parameter is a character constant containing the class of the method. The third parameter is a character constant containing the method name. The special method name *CONSTRUCTOR means that the method is a constructor; this method can be used to instantiate a class (create a new class instance).

For more information about invoking Java procedures, see *ILE RPG Programmer's Guide*.

EXTPROC(name)
Specifies an external procedure.

The EXTPROC keyword indicates the external name of the procedure whose prototype is being defined. The name can be a character constant or a procedure pointer. When EXTPROC is specified, then the procedure should be called using CALLB or CALLP.

If EXTPROC is not specified, then the compiler assumes that you are defining a procedure, and assigns it the external name found in positions 7-21.

If the name specified for EXTPROC (or the prototype name, if EXTPROC is not specified) starts with ″CEE″ or an underscore ('_'), the compiler will treat this as a

system built-in. If it is not actually a system built-in, then a warning will appear in the listing; To avoid confusion with system provided APIs, you should not name your procedures starting with "CEE".

If a procedure pointer is specified, it must be assigned a valid address before it is used in a call. It should point to a procedure whose return value and parameters are consistent with the prototype definition.

For example, to define the prototype for the procedure SQLAllocEnv, that is in the program QSQCLI, the following definition specification could be coded:

```
      D SQLEnv          PR                  EXTPROC('SQLAllocEnv')
```

Figure 91 shows an example of the EXTPROC keyword with a procedure pointer as its parameter.

```
D* Assume you are calling a procedure that has a procedure
D* pointer as the EXTPROC.  Here is how the prototype would
D* be defined:
D*
D DspMsg          PR            10A   EXTPROC(DspMsgPPtr)
D  Msg                         32767A
D  Length                          4B 0 VALUE
D*
D* Here is how you would define the prototype for a procedure
D* that DspMsgPPtr could be assigned to.
D*
D MyDspMsg        PR                  LIKE(DspMsg)
D  Msg                         32767A
D  Length                          4B 0 VALUE
C*
C* Before calling DSPMSG, you would assign DSPMSGPPTR
C* to the actual procedure name of MyDspMsg, that is
C* MYDSPMSG.
C*
C                    EVAL      DspMsgPPtr = %paddr('MYDSPMSG')
C                    EVAL      Reply = DspMsg(Msg, %size(Msg))
 ...
P MyDspMsg       B
```

Figure 91. Specifying the External Name of a Prototyped Procedure

The extended form of the EXTPROC keyword can be used to prototype Java methods that are called from VARPG Java applications. See "Prototyping Java Methods."

Prototyping Java Methods

Java methods must be prototyped so that VARPG Java applications can call them correctly. The compiler must know the name of the method, the class it belongs to, the data types of the parameters and the data type of the returned value (if any), and whether or not the method is a static method.

Use the extended form of the EXTPROC keyword to specify the name of the method and the class it belongs to. The format of the EXTPROC keyword is:

```
EXTPROC(*JAVA:class_name:method_name | *JAVARPG:class_name:method_name)
```

The possible parameter values are:

***JAVA:class_name:method_name**
Identifies the method as a Java method that was generated from code originally written in Java.

***JAVARPG:class_name:method_name**
Identifies a VARPG-generated Java method.

VARPG-generated methods allow certain data types to be passed by reference that normally cannot be passed by reference in Java. This allows you to use the same source code when targeting for Windows applications or when generating Java source code.

The class and method names must be character literals, and are case sensitive. The class name must be a fully qualified Java class name. The method name must be the name of the method to be called.

The data types of the parameters and the returned value of the method are specified in the same way as they are when prototyping a subprocedure. However, note that the compiler maps VARPG data types to Java data types as follows:

Java Data Type	VARPG Data Type
boolean	indicator (N)
byte[]	alpha (A of any length)
byte	integer (3I)
int	integer (10I)
short	integer (5I)
long	integer (20I)
float	float (4F)
double	float (8F)
any object	object (O)

When calling VARPG-generated methods (*JAVARPG), you can specify the Packed, Zoned, Binary, or Unsigned data type as the data type of parameters and returned values. Methods generated from Java source code cannot use these data types on the prototype for parameters or return values.

When calling a method, the compiler will accept arrays as parameters if the parameter is prototyped using the DIM keyword. Otherwise, only scalar fields, data structures, and tables will be accepted.

You cannot call methods that expect the Java **char** data type or return this value type.

If the return value of a method is an object, provide the class of the object by coding the CLASS keyword on the prototype. The class name specified will be that of the object being returned.

If the method being called is a static method, specify the STATIC keyword on the prototype.

In Java, the following data types can only be passed by value:

```
byte
int
short
long
float
double
```

Specify the VALUE keyword on the prototype for parameters of these types. The VALUE keyword is not required when calling VARPG-generated methods as these data types can be passed by reference.

Objects can only be passed by reference. The VALUE keyword cannot be specified with type 'O'. Since arrays are seen by Java as objects, parameters mapping to arrays must also be passed by reference. This includes byte arrays.

For more information on calling Java methods and examples, see the *Programming with VisualAge RPG* manual.

FROMFILE(file_name)

The FROMFILE keyword specifies the file with input data for the pre-runtime array or table being defined. The FROMFILE keyword must be specified for every pre-runtime array or table used in the program.

See "TOFILE(file_name)" on page 295.

INZ{(initial value)}

The INZ keyword initializes the standalone field, data structure or data-structure subfield to the default value for its data type or, optionally, to the constant specified in parentheses.

- For a program described data structure, no parameter is allowed for the INZ keyword.
- For an externally described data structure, only the *EXTDFT parameter is allowed.
- For a data structure that is defined with the LIKEDS keyword, the value *LIKEDS specifies that subfields are initialized in the same way as the parent data structure. This applies only to initialization specified by the INZ keyword on the parent subfield. It does not apply to initialization specified by the CTDATA or FROMFILE keywords. If the parent data structure has some subfields initialized by CTDATA or FROMFILE, the data structure initialized with INZ(*LIKEDS) will not have the CTDATA or FROMFILE data.
- For an object, only the *NULL parameter is allowed. Every object is initialized to *NULL, whether or not you specify INZ(*NULL).

The initial value specified must be consistent with the type being initialized. The initial value can be a literal, named constant, figurative constant, built-in function, or one of the special values *EXTDFT, *LIKEDS, or *NULL. When initializing Date or Time data type fields or named constants with Date or Time values, the format of the literal must be consistent with the default format as derived from the Control specification, regardless of the actual format of the date or time field.

A numeric field may be initialized with any type of numeric literal. However, a float literal can only be used with a float field. Any numeric field can be initialized with a hexadecimal literal of 16 digits or fewer. In this case, the hexadecimal literal is considered an unsigned numeric value.

Specifying INZ(*EXTDFT) initializes externally described data-structure subfields with the default values from the DFT keyword in the DDS. If no DFT or constant value is specified, the DDS default value for the field type is used. You can override the value specified in the DDS by coding INZ with or without a parameter on the subfield specification.

Specifying INZ(*EXTDFT) on the external data structure definition, initializes all externally described subfields to their DDS default values. If the externally described data structure has additional program described subfields, these are initialized to the RPG default values.

When using INZ(*EXTDFT), take note of the following:
- If the DDS value for a date or time field is not in the RPG internal format, the value will be converted to the internal format in effect for the program.
- External descriptions must be in physical files.
- If *NULL is specified for a null-capable field in the DDS, the compiler will use the DDS default value for that field as the initial value.
- If DFT('') is specified for a varying length field, the field will be initialized with a string of length 0.
- INZ(*EXTDFT) is not allowed if the CVTOPT option is in effect.
- If no initial value or *NULL is specified for date, time, or timestamp fields, the initial value for the field is set to *LOVAL.

Please see "Initialization of Nested Data Structures" on page 161 for a complete description of the use of the INZ keyword in the inititlization of nested data structures.

A data structure, data-structure subfield, or standalone field defined with the INZ keyword cannot be specified as a parameter on an *ENTRY PLIST.

Note: When the INZ parameter is not specified:
- Static standalone fields and subfields of initialized data structures are initialized to their default initial values (for example, blanks for character, 0 for numeric).
- Subfields of static uninitialized data structures (INZ not specified on the definition specification for the data structure) are initialized to blanks (regardless of their data type).
- Fields in automatic storage are not initialized.

This keyword is not valid in combination with BASED.

LIKE(RPG_name)

The LIKE keyword is used to define an item like an existing one. When the LIKE keyword is specified, the item being defined takes on the length and data format of the item specified as the parameter. Standalone fields prototypes, parameters, and data-structure subfields may be defined using this keyword. The parameter of LIKE can be a standalone field, a data structure, a data structure subfield, a parameter in a procedure interface definition, or a prototype name. The data type entry (position 40) must be blank.

This keyword is similar to the *LIKE DEFINE operation code (see "Defining a Field Based on Another Field" on page 548). However, it differs from *LIKE DEFINE in that the defined data takes on the data format and CCSID as well as the length.

Note: Attributes such as NOOPT, ASCEND, CONST and null capability are not inherited from the parameter of LIKE by the item defined. Only the data type, length, decimal positions, and CCSID are inherited.

If the parameter of LIKE is a prototype, then the item being defined will have the same data type as the return value of the prototype. If there is no return value, then an error message is issued.

This keyword can be used when defining CALLP parameters. See "Defining a Field Based on Another Field" on page 548.

LIKE can be used to define character fields, graphic fields, graphic characters, numeric fields, and arrays. Here are some considerations for using the LIKE keyword with different data types:

- **For character fields**, the number specified in the To/Length entry is the number of additional (or fewer) characters
- **For graphic or UCS-2 fields**, the number specified in the To/Length entry is the number of additional (or fewer) graphic or UCS-2 characters (1 graphic or UCS-2 character = 2 bytes).
- **For numeric fields**, the number specified in the To/Length entry is the number of additional (or fewer) digits. For integer or unsigned fields, adjustment values must be such that the resulting number of digits for the field are 3, 5, 10, or 20. For float fields, length adjustment is not allowed.
- **For date, time, timestamp, basing pointer, or procedure pointer fields**, the To/Length entry (positions 33-39) must be blank.

When LIKE is used to define an array, the DIM keyword is still required to define the array dimensions. However, DIM(%elem(array)) can be used to define an array exactly like another array.

Use LIKEDS to define a data structure like another data structure, with the same subfields.

The following are examples of defining data using the LIKE keyword.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D.................................Keywords++++++++++++++++++++++++++++++
D*
D*  Define a field like another with a length increase of 5 characters.
D*
D  Name           S             20
D  Long_name      S             +5    LIKE(Name)
D*
D*  Define a data structure subfield array with DIM(20) like another
D*  field and initialize each array element with the value *ALL'X'.
D*  Also, declare another subfield of type pointer immediately
D*  following the first subfield. Pointer is implicitly defined
D*  with a length of 4 bytes.
D*
D  Struct         DS
D   Dim20                              LIKE(Name) DIM(20) INZ(*ALL'X')
D   Pointer                      *
```

Figure 92. Defining fields LIKE other fields

LIKE(object-name)

You can use the LIKE keyword to specify that one object has the same class as a previously defined object. Only the values on the CLASS keyword are inherited.

```
* Variables MyString and OtherString are both Java String objects.
D MyString        S               O   CLASS(*JAVA
D                                         :'java.lang.String')
D OtherString     S                   LIKE(MyString)
 * Proc is a Java method returning a Java String object
D Proc            PR                  EXTPROC(*JAVA:'MyClass':'meth')
D                                     LIKE(MyString)
```

Figure 93. Defining objects LIKE other objects

Note: You cannot use the *LIKE DEFINE operation to define an object. You must use the LIKE keyword.

LIKEDS(data_structure_name)

The LIKEDS keyword is used to define a data structure, data structure subfield, prototyped return value, or prototyped parameter like another data structure. The subfields of the new item will be identical to the subfields of the other data structure.

The names of the subfields will be qualified with the new data structure name. An unqualified subfield named *subfield* or a qualified subfield named *dsname.subfield* will result in a new subfield named *newdsname.subfield*. An unnamed subfield will also have no name in the new data structure.

LIKEDS can be coded for subfields of a qualified data structure. When LIKEDS is coded on a data structure subfield definition, the subfield data structure is automatically defined as QUALIFIED. Subfields in a LIKEDS subfield data structure are referenced in fully qualified form: `"ds.subf.subfa"`. Subfields defined with LIKEDS are themselves data structures, and can be used wherever a data structure is required.

The value of the ALIGN keyword are inherited by the new data structure. The values of the OCCURS, DIM, NOOPT, and INZ keywords are not inherited. To initialize the subfields in the same way as the parent data structure, specify INZ(*LIKEDS).

```
D sysName         DS                  qualified
D   lib                         10A   inz('*LIBL')
D   obj                         10A
D userSpace       DS                  LIKEDS(sysName) INZ(*LIKEDS)
 // The variable "userSpace" was initialized with *LIKEDS, so the
 // first 'lib' subfield was initialized to '*LIBL'.  The second
 // 'obj' subfield must be set using a calculation.
C                   eval      userSpace.obj = 'TEMPSPACE'
```

*Figure 94. Using INZ(*LIKEDS)*

```
P createSpace     B
D createSpace     PI
D  name                               LIKEDS(sysName)
 /free
     if name.lib = *blanks;
          name.lib = '*LIBL';
     endif;
     QUSCRTUS (name : *blanks : 4096 : ' ' : '*USE' : *blanks);
 /end-free
P createSpace     E
```

Figure 95. Using a data structure parameter in a subprocedure

LIKEREC(intrecname{:*ALL|*INPUT|*OUTPUT |*KEY})

Keyword LIKEREC is used to define a data structure, data structure subfield, prototyped return value, or prototyped parameter like a record. The subfields of the data structure will be identical to the fields in the record. LIKEREC can take an optional second parameter which indicates which fields of the record to include in the data structure. These include:

- ***ALL** All fields in the external record are extracted.
- ***INPUT** All input-capable fields are extracted. (This is the default.)
- ***OUTPUT** All output-capable fields are extracted.
- ***KEY** The key fields are extracted in the order that the keys are defined on the K specification in the DDS.

The following should be taken into account when using the LIKEREC keyword:

- The first parameter for keyword LIKEREC is a record name in the program. If the record name has been renamed, it is the internal name for the record.
- The second parameter for LIKEREC must match the definition of the associated record or file. *INPUT is only allowed for input and update capable records; *OUTPUT is only allowed for output capable records; *ALL is allowed for any type of record; and *KEY is only allowed for keyed files. If not specified, the parameter defaults to *INPUT.
- For *INPUT and *OUTPUT, subfields included in the data structure occupy the same start positions as in the external record description.
- If a prefix was specified for the file, the specified prefix is applied to the names of the subfields.
- Even if a field in the record is explicitly renamed on an input specification the external name (possibly prefixed) is used, not the internal name.

- A data structure defined with LIKEREC is a QUALIFIED data structure. The names of the subfields will be qualified with the new data structure name, DS1.SUBF1.
- LIKEREC can be coded for subfields of a qualified data structure. When LIKEREC is coded on a data structure subfield definition, the subfield data structure is automatically defined as QUALIFIED. Subfields in a LIKEREC subfield data structure are referenced in fully qualified form: `"ds.subf.subfa"`. Subfields defined with LIKEREC are themselves data structures, and can be used wherever a data structure is required.

LINKAGE(linkage_type)

When you define a program name to be used with the CALL and START operations, the LINKAGE keyword specifies the location of the called program. Use the *SERVER parameter value with this keyword for the CALL operation. The *SERVER parameter specifies that the program which you are calling exists on an iSeries server. Use the *CLIENT parameter value with this keyword for the START operation.

Specify LINKAGE(*SERVER) on a prototype definition for a remote program on an iSeries server.

The LINKAGE keyword, together with the DLL keyword, specifies the Linkage convention (interface) to be used when invoking functions in a dynamic-link library (DLL). The linkage convention specified must match that of the entry point in the external DLL that is to be accessed. Windows System APIs use the StdCall linkage convention. So, when prototyping a Windows System API, specify LINKAGE(*STDCALL).

Do not use the LINKAGE keyword if you use the DLL keyword to prototype a VARPG subprocedure in a NOMAIN application. The compiler will use the default __cdecl linkage convention.

When protoyping your own DLLs, create them with the __stdcall or __cdecl linkage convention. Using other linkage conventions may cause unpredictable results or runtime errors.

MSGDATA(msgdata1:msgdata2....)

The MSGDATA keyword defines the substitution text, used in Factor 1 of the DSPLY operation code, in the form of a list of field names that you define in your program. The VisualAge RPG compiler replaces each substitution variable with the corresponding field defined. For example, %1 would be replaced by the first field defined in MSGDATA, %2 by the second field defined in MSGDATA, and so on. Substitution variables are defined by entering the percent (%) character followed by a single digit (1 to 9). You can specify a maximum of 3 parameters per keyword.

The MSGDATA and MSGNBR keywords are used together.

MSGNBR(*MSGnnnn or fieldname)

The MSGNBR keyword defines the message number used in Factor 1 of the DSPLY operation code. The message number can be a maximum of 4 digits in length. You must specify one of the following:
- The message identifier (for example, *MSG0001)
- A field containing the message number(for example, *MSG0001)

If you have substitution text in your messages, use the MSGNBR and MSGDATA keywords together.

MSGTEXT('message text')

The MSGTEXT keyword defines the message text, which is contained within single quotes ('). This text is used in Factor 1 of the DSPLY operation code. This keyword cannot be used if the BUTTON, MSGDATA, MSGNBR, MSGTITLE, or STYLE keywords are used.

MSGTITLE('title text')

The MSGTITLE keyword specifies the title text for the message window (Factor 2 of the DSPLY operation code).You can enter an 8-character message identifier enclosed in single quotes ('), for example, '*MSG0001', or a 4-digit message number. If you use a message number, the text is retrieved from the message file. (Use the Define messages option of the GUI designer to specify titles in message format.)

This keyword cannot be used if the MSGDATA, MSGNBR, or MSGTEXT keywords are used.

NOOPT

No optimization is to be performed on the standalone field, parameter, or data structure for which this keyword is specified. This insures that the content of the data item is the latest assigned value. This may be necessary for those fields whose values are used in exception handling.

Note: The optimizer may keep some values in registers and restore them only to storage at predefined points during normal program execution. Exception handling may break this normal execution sequence, and consequently program variables contained in registers may not be returned to their assigned storage locations. As a result, when those variables are used in exception handling, they may not contain the latest assigned value. The NOOPT keyword ensures their currency.

If a data item which is to be passed by reference is defined with the NOOPT keyword, then any prototype or procedure interface parameter definition must also have the NOOPT keyword specified. This requirement does not apply to parameters passed by value.

All keywords allowed for standalone field definitions, parameters, or data structure definitions are allowed with NOOPT.

This keyword can be used when defining CALLP parameters.

NOWAIT

The NOWAIT keyword allows you to call an OS/400 program that uses a workstation file. See "Calling an OS/400 Program that Uses a Workstation File" on page 518 for details.

OCCURS(numeric_constant)

The OCCURS keyword specifies the number of occurrences of a multiple occurrence data structure.

The numeric_constant parameter must be a value greater than 0 with no decimal positions. It can be a numeric literal, a built-in function returning a numeric value, or a numeric constant.

The constant value does not need to be known at the time the keyword is processed, but the value must be known at compile-time.

This keyword is not valid for a program status data structure, a file information data structure, or a data area data structure.

If a multiple occurrence data structure contains pointer subfields, the distance between occurrences must be an exact multiple of 4 because of system storage restrictions for pointers. This means that the distance between occurrences may be greater than the length of each occurrence.

The following example shows the storage allocation of a multiple occurrence data structure with pointer subfields.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D DS1             DS                  OCCURS(2)
D   POINTER                       4*
D   FLD5                          5
D DS2             DS                  OCCURS(2)
D   CHAR16                        4
D   CHR5                          5
```

Allocation of fields in storage. The occurrences of DS1 are 8 bytes apart, while the occurrences of DS2 are 9 bytes apart.

DS1 OCCURRENCE 1			DS1 OCCURRENCE 2		
POINTER	FLD5	(fill)	POINTER	FLD5	(fill)

DS2 OCCURRENCE 1		DS2 OCCURRENCE 2	
CHAR4	CHR5	CHAR4	CHR5

Figure 96. Storage Allocation of Multiple Occurrence Data Structure with Pointer Subfields

OPTIONS(*OMIT *VARSIZE *STRING *TRIM *RIGHTADJ)

The OPTIONS keyword is used to specify:

- Whether the special value *OMIT can be passed for the parameter passed by reference
- Whether a parameter that is passed by reference can be shorter in length than is specified in the prototype.
- Whether the called program or procedure is expecting a pointer to a null-terminated string, allowing you to specify a character expression as the passed parameter.
- Whether the parameter should be trimmed of blanks before being passed.
- Whether the parameter value should be right-adjusted in the passed parameter.

The OPTIONS keyword cannot be specified without a parameter. More than one parameter may be specified on one definition specification, but each parameter must be different.

The value can be *OMIT, *STRING, *VARSIZE, or *RIGHTADJ.

When OPTIONS(*OMIT) is specified, the value *OMIT is allowed for that parameter. *OMIT is only allowed for CONST parameters and parameters which are passed by reference.

OPTIONS(*VARSIZE) is valid only for parameters passed by reference that have a character, graphic, or UCS-2 data type, or that represent an array of any type.

When OPTIONS(*VARSIZE) is specified, the passed parameter may be shorter or longer in length than is defined in the prototype. It is then up to the called program or subprocedure to ensure that it accesses only as much data as was passed. To communicate the amount of data passed, you can either pass an extra parameter containing the length, or use operational descriptors for the subprocedure. For variable-length fields, you can use the %LEN built-in function to determine the current length of the passed parameter.

When OPTIONS(*VARSIZE) is omitted for fixed-length fields, you must pass *at least* as much data as is required by the prototype; for variable-length fields, the parameter must have the same declared maximum length as indicated on the definition.

Note: For the parameter passing options *OMIT and *VARSIZE, it is up to the programmer of the procedure to ensure that these options are handled.

When OPTIONS(*STRING) is specified for a basing pointer parameter passed by value or by constant-reference, you may either pass a pointer or a character expression. If you pass a character expression, a temporary value will be created containing the value of the character expression followed by a null-terminator (x'00'). The address of this temporary value will be passed to the called program or procedure.

When OPTIONS(*RIGHTADJ) is specified for a CONST or VALUE parameter in a prototype, the character, graphic, or UCS-2 parameter value is right adjusted. This keyword is not allowed for a varying length parameter within a procedure prototype. Varying length values may be passed as parameters on a procedure call where the corresponding parameter is defined with OPTIONS(*RIGHTADJ).

When OPTIONS(*TRIM) is specified for a CONST or VALUE parameter of type character, UCS-2 or graphic, the passed parameter is copied without leading and trailing blanks to a temporary. If the parameter is not a varying length parameter, the trimmed value is padded with blanks (on the left if OPTIONS(*RIGHTADJ) is specified, otherwise on the right). Then the temporary is passed instead of the original parameter. Specifying OPTIONS(*TRIM) causes the parameter to be passed exactly as though %TRIM were coded on every call to the procedure.

When OPTIONS(*STRING : *TRIM) is specified for a CONST or VALUE parameter of type pointer, the character parameter or %STR of the pointer parameter is copied without leading or trailing blanks to a temporary, a null-terminator is added to the temporary and the address of the temporary is passed.

You can specify more than one option. For example, OPTIONS(*STRING : *TRIM).

The following example shows how to code a prototype and procedure using OPTIONS(*OMIT) to indicate that the special value *OMIT may be passed as a parameter.

```
F*
FQSYSPRT   O    F   10        PRINTER USROPN
D*
D* The following prototype describes a procedure that allows
D* the special value *OMIT to be passed as a parameter.
D* If the parameter is passed, it is set to '1' if an error
D* occurred, and '0' otherwise.
D OpenFile        PR
D   Error                       1A   OPTIONS(*OMIT)
C*
C                   SETOFF                                       10
C* The first call to OpenFile assumes that no error will occur,
C* so it does not bother with the error code and passes *OMIT.
C                   CALLP     OpenFile(*OMIT)
C*
C* The second call to OpenFile passes an indicator so that
C* it can check whether an error occurred.
C                   CALLP     OpenFile(*IN10)
C                   IF        *IN10
C                   ... an error occurred
C                   ENDIF
C                   RETURN
P*---------------------------------------------------------------
P* OpenFile
P* This procedure must check the number of parameters.
P*---------------------------------------------------------------
P OpenFile        B
D OpenFile        PI
D   Error                       1A   OPTIONS(*OMIT)
D SaveIn01        S               1A
C* Save the current value of indicator 01 in case it is being
C* used elsewhere.
C                   EVAL      SaveIn01 = *IN01
C* Open the file.  *IN01 will indicate if an error occurs.
C                   OPEN      QSYSPRT                             01
C* If the Error parameter was passed, update it with the indicator
C                   IF        %ADDR(Error) <> *NULL
C                   EVAL      Error = *IN01
C                   ENDIF
C* Restore *IN01 to its original value.
C                   EVAL      *IN01 = SaveIn01
P OpenFile        E
```

*Figure 97. Using OPTIONS(*OMIT)*

The following example shows how to code a prototype and procedure allowing variable-length parameters, using OPTIONS(*VARSIZE).

```
D* The following prototype describes a procedure that allows
D* both a variable-length array and a variable-length character
D* field to be passed.  Other parameters indicate the lengths.
D Search          PR             5U 0
D   SearchIn                    50A   OPTIONS(*VARSIZE)
D                                     DIM(100) CONST
D   ArrayLen                     5U 0 VALUE
D   ArrayDim                     5U 0 VALUE
D   SearchFor                   50A   OPTIONS(*VARSIZE) CONST
D   FieldLen                     5U 0 VALUE
D*
D Arr1            S               1A   DIM(7) CTDATA PERRCD(7)
D Arr2            S              10A   DIM(3) CTDATA
D Elem            S               5U 0
C* Call Search to search an array of 7 elements of length 1 with
C* a search argument of length 1.  Since the '*' is in the 5th
C* element of the array, Elem will have the value 5.
C                   EVAL      Elem = Search(Arr1 :
C                                           %SIZE(Arr1) : %ELEM(Arr1) :
C                                           '*' : 1)
C* Call Search to search an array of 3 elements of length 10 with
C* a search argument of length 4.  Since 'Pink' is not in the
C* array, Elem will have the value 0.
C                   EVAL      Elem = Search(Arr2 :
C                                           %SIZE(Arr2) : %ELEM(Arr2) :
C                                           'Pink' : 4)
C                   RETURN
```

*Figure 98. Using OPTIONS(*VARSIZE) (Part 1 of 2)*

```
P*-------------------------------------------------------------
P* Search:
P*   Searches for SearchFor in the array SearchIn.  Returns
P*   the element where the value is found, or 0 if not found.
P*   The character parameters can be of any length or
P*   dimension since OPTIONS(*VARSIZE) is specified for both.
P*-------------------------------------------------------------
P Search          B
D Search          PI             5U 0
D   SearchIn                    50A   OPTIONS(*VARSIZE)
D                                     DIM(100) CONST
D   ArrayLen                     5U 0 VALUE
D   ArrayDim                     5U 0 VALUE
D   SearchFor                   50A   OPTIONS(*VARSIZE) CONST
D   FieldLen                     5U 0 VALUE
D I               S              5U 0
C* Check each element of the array to see if it the same
C* as the SearchFor.  Use the dimension that was passed as
C* a parameter rather than the declared dimension.  Use
C* %SUBST with the length parameter since the parameters may
C* not have the declared length.
C     1             DO        ArrayDim      I                 5 0
C* If this element matches SearchFor, return the index.
C                   IF        %SUBST(SearchIn(I) : 1 : ArrayLen)
C                             = %SUBST(SearchFor : 1 : FieldLen)
C                   RETURN    I
C                   ENDIF
C                   ENDDO
C* No matching element was found.
C                   RETURN    0
P Search          E

Compile-time data section:

     **CTDATA ARR1
     A2$@*jM
     **CTDATA ARR2
     Red
     Blue
     Yellow
```

*Figure 98. Using OPTIONS(*VARSIZE) (Part 2 of 2)*

The following example shows how to use OPTIONS(*STRING) to code a prototype and procedure that use a null-terminated string parameter.

```
D* The following prototype describes a procedure that expects
D* a null-terminated string parameter.  It returns the length
D* of the string.
D StringLen       PR             5U 0
D   Pointer                       *   VALUE OPTIONS(*STRING)
D P               S               *
D Len             S              5U 0
C*
C* Call StringLen with a character literal.  The result will be
C* 4 since the literal is 4 bytes long.
C                   EVAL      Len = StringLen('abcd')
C*
C* Call StringLen with a pointer to a string.  Use ALLOC to get
C* storage for the pointer, and use %STR to initialize the storage
C* to 'My string¬' where '¬' represents the null-termination
C* character x'00'.
C* The result will be 9 which is the length of 'My string'.
C                   ALLOC     25            P
C                   EVAL      %STR(P:25) = 'My string'
C                   EVAL      Len = StringLen(P)
C* Free the storage.
C                   DEALLOC                 P
C                   RETURN
P*-------------------------------------------------------------
P* StringLen:
P*   Returns the length of the string that the parameter is
P*   pointing to.
P*-------------------------------------------------------------
P StringLen       B
D StringLen       PI             5U 0
D   Pointer                       *   VALUE OPTIONS(*STRING)
C                   RETURN    %LEN(%STR(Pointer))
P StringLen       E
```

*Figure 99. Using OPTIONS(*STRING)*

```
* The following prototype describes a procedure that expects
* these parameters:
*   1. trimLeftAdj   - a fixed length parameter with the
*                      non-blank data left-adjusted
*   2. leftAdj       - a fixed length parameter with the
*                      value left-adjusted (possibly with
*                      leading blanks)
*   3. trimRightAdj  - a fixed length parameter with the
*                      non-blank data right-adjusted
*   4. rightAdj      - a fixed length parameter with the
*                      value right-adjusted (possibly with
*                      trailing blanks)
*   5. trimVar       - a varying parameter with no leading
*                      or trailing blanks
*   6. var           - a varying parameter, possibly with
*                      leading or trailing blanks
D trimProc        PR
D   trimLeftAdj                10a   const options(*trim)
D   leftAdj                    10a   const
D   trimRightAdj               10a   value options(*rightadj : *trim)
D   rightAdj                   10a   value options(*rightadj)
D   trimVar                    10a   const varying options(*trim)
D   var                        10a   value varying
* The following prototype describes a procedure that expects
* these parameters:
*   1. trimString    - a pointer to a null-terminated string
*                      with no leading or trailing blanks
*   2. string        - a pointer to a null-terminated string,
*                      possibly with leading or trailing blanks
D trimStringProc  PR
D   trimString                   *   value options(*string : *trim)
D   string                       *   value options(*string)
D ptr             s              *
 /free
       // trimProc is called with the same value passed
       // for every parameter
       //
       // The called procedure receives the following parameters
       //   trimLeftAdj    'abc       '
       //   leftAdj        ' abc      '
       //   trimRightAdj   '       abc'
       //   rightAdj       '      abc '
       //   trimVar        'abc'
       //   var            ' abc '

       callp trimProc (' abc ' : ' abc ' : ' abc ' :
                                ' abc ' : ' abc ' : ' abc ' );
       // trimStringProc is called with the same value passed
       // for both parameters
       //
       // The called procedure receives the following parameters,
       // where ¬ represents x'00'
       //   trimstring     pointer to 'abc¬'
       //   string         pointer to ' abc ¬'

       callp trimStringProc (' abc ' : ' abc ');

       // trimStringProc is called with the same pointer passed
       // to both parameters
       //
       // The called procedure receives the following parameters,
       // where ¬ represents x'00'
       //   trimstring     pointer to 'xyz¬'
       //   string

       pointer to ' xyz ¬'
       ptr = %alloc (6);
       %str(ptr : 6) = ' xyz ';
       callp trimStringProc (ptr : ptr);
```

*Figure 100. Using OPTIONS(*TRIM)*

OVERLAY(name{:pos | *NEXT})

The OVERLAY keyword overlays the storage of one subfield with that of another subfield, or with that of the data structure itself. This keyword is allowed only for data structure subfields.

The Name-entry subfield overlays the storage specified by the name parameter at the position specified by the pos parameter. If pos is not specified, it defaults to 1.

Note: The pos parameter is in units of bytes, regardless of the types of the subfields.

Specifying OVERLAY(name:*NEXT) positions the subfield at the next available position within the overlaid field. (This will be the first byte past all other subfields prior to this subfield that overlay the same subfield.)

The following rules apply to OVERLAY keyword:

- The name parameter must be the name of a subfield defined previously in the current data structure, or the name of the current data structure.
- If the data structure is qualified, the first parameter to the OVERLAY keyword must be specified without the qualifying data structure name. In the following example, subfield MsgInfo.MsgPrefix overlays subfield MsgInfo.MsgId.

```
D MsgInfo         DS                  QUALIFIED
D   MsgId                        7
D     MsgPrefix                  3    OVERLAY(MsgId)
```

- The pos parameter (if specified) must be a value greater than 0 with no decimal positions. It can be a numeric literal, a built-in function returning a numeric value, or a numeric constant. If pos is a named constant, it must be defined prior to this specification.
- The OVERLAY keyword is not allowed when the From-Position entry is not blank.
- If the name parameter is a subfield, the subfield being defined must be contained completely within the subfield specified by the name parameter.
- The subfield being defined must be contained completely within the subfield specified by the name parameter.
- Alignment of subfields defined using the OVERLAY keyword must be done manually. If they are not correctly aligned, a warning message is issued.
- If the subfield specified as the first parameter for the OVERLAY keyword is an array, the OVERLAY keyword applies to each element of the array. That is, the field being defined is defined as an array with the same number of elements. The first element of this array overlays the first element of the overlayed array, the 2nd element of this array overlays the 2nd element of the overlayed array, etc. No array keywords may be specified for the subfield with the OVERLAY keyword in this situation. See Figure 101 on page 292 and "SORTA (Sort an Array)" on page 686.

  If the subfield name, specified as the first parameter for the OVERLAY keyword, is an array and its element length is longer than the length of the subfield being defined, the array elements of the subfield being defined are not stored contiguously. Such an array is not allowed as the Result Field of a PARM operation or in Factor 2 or the Result Field of a MOVEA operation.
- If the ALIGN keyword is specified for the data structure, subfields defined with OVERLAY(name:*NEXT) are aligned to their preferred alignment. Pointer subfields are always aligned on a 4-byte boundary.
- If a subfield with overlaying subfields is not otherwise defined, the subfield is implicitly defined as follows:
  - The start position is the first available position in the data structure.

– The length is the minimum length that can contain all overlaying subfields. If the subfield is defined as an array, the length will be increased to ensure proper alignment of all overlaying subfields.

Examples

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D DataStruct      DS
D   A                              10    DIM(5)
D     B                             5    OVERLAY(A)
D     C                             5    OVERLAY(A:6)
```

Allocation of fields in storage:

A(1)		A(2)		A(3)		A(4)		A(5)	
B(1)	C(1)	B(2)	C(2)	B(3)	C(3)	B(4)	C(4)	B(5)	C(5)

Figure 101. Storage Allocation of Subfields with Keywords DIM and OVERLAY

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D DataStruct      DS
D   A                              5
D     B                            1    OVERLAY(A) DIM(4)
```

Allocation of fields in storage:

A				
B(1)	B(2)	B(3)	B(4)	

Figure 102. Storage Allocation of Subfields with Keywords DIM and OVERLAY

The following example shows two equivalent ways of defining subfield overlay positions: explicitly with (name:pos) and implicitly with (name:*NEXT).

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
 * Define subfield overlay positions explicitly
D DataStruct      DS
D   PartNumber                   10A
D     Family                      3A   OVERLAY(PartNumber)
D     Sequence                    6A   OVERLAY(PartNumber:4)
D     Language                    1A   OVERLAY(PartNumber:10)
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
 * Define subfield overlay positions with *NEXT
D DataStruct      DS
D   PartNumber
D     Family                      3A   OVERLAY(PartNumber)
D     Sequence                    6A   OVERLAY(PartNumber:*NEXT)
D     Language                    1A   OVERLAY(PartNumber:*NEXT)
```

*Figure 103. Defining Subfield Overlay Positions with *NEXT*

PACKEVEN

The PACKEVEN keyword indicates that the packed field or array has an even number of digits. The keyword is only valid for packed program-described data-structure subfields defined using FROM/TO positions. For a field or array element of length N, if the PACKEVEN keyword is not specified, the number of digits is 2N - 1; if the PACKEVEN keyword is specified, the number of digits is 2(N-1).

PERRCD(numeric_constant)

The PERRCD keyword specifies the number of elements per record for a compile-time or a pre-runtime array or table. If the PERRCD keyword is not specified, the number of elements per record defaults to one (1).

The numeric_constant parameter must be a value greater than 0 with no decimal positions. It can be a numeric literal, a built-in function returning a numeric value, or a numeric constant. If the parameter is a named constant, it does not need to be defined prior to this specification.

The PERRCD keyword is valid only when the keyword FROMFILE, TOFILE, or CTDATA is specified.

PREFIX(prefix{:nbr_of_char_replaced})

The PREFIX keyword allows the specification of a character string or character literal which is to be prefixed to the subfield names of the externally described data structure being defined. In addition, you can optionally specify a numeric value to indicate the number of characters, if any, in the existing name to be replaced. If the parameter 'nbr_of_char_replaced' is not specified, then the string is attached to the beginning of the name.

If the 'nbr_of_char_replaced' is specified, it must represent a numeric value between 0 and 9 with no decimal places. Specifying a value of zero is the same as not specifying 'nbr_of_char_replaced' at all. For example, the specification PREFIX(YE:3) would change the field name 'YTDTOTAL' to 'YETOTAL'.

The following rules apply:

- Subfields that are explicitly renamed using the EXTFLD keyword are not affected by this keyword.
- The total length of a name after applying the prefix must not exceed the maximum length of an RPG field name.
- If the number of characters in the name to be prefixed is less than or equal to the value represented by the 'nbr_of_char_replaced' parameter, then the entire name is replaced by the prefix_string.
- The prefix cannot end in a period.
- If the prefix is a character literal, it must be uppercase.

PROCPTR

The PROCPTR keyword defines an item as a procedure pointer. The Internal Data-Type field (position 40) must contain a *.

QUALIFIED

The QUALIFIED keyword specifies that the subfields of a data structure will be accessed by specifying the data structure name followed by a period and the subfield name. The data structure must have a name.

The subfields can have any valid name, even if the name has been used elsewhere in the program. This is illustrated in the following example:

```
 * In this example, FILE1 and FILE2 are the names of files.  FILE1 and FILE2 are
 * also subfields of qualified data structure FILESTATUS.  This is valid,
 * because the subfields FILE1 and FILE2 must be qualified by the data structure
 * name: FILESTATUS.FILE1 and FILESTATUS.FILE2.
Ffile1     if   e             disk         remote
Ffile2     if   e             disk          remote

D fileStatus      ds                  qualified
D   file1                         N
D   file2                         N

C                   open(e)   file1
C                   eval      fileStatus.file1 = %error
```

STATIC

The STATIC keyword is used:
- To specify that a local variable is stored in static storage
- To specify that a Java method is defined as a static method.

For a local variable of a subprocedure, the STATIC keyword specifies that the data item is to be stored in static storage, and thereby hold its value across calls to the procedure in which it is defined. The keyword can only be used within a subprocedure. All global fields are static.

The data item is initialized when the subprocedure it is contained in is first activated. It is *not* reinitialized again, even if the subprocedure is called again.

If STATIC is not specified, then any locally-defined data item is stored in automatic storage. Data stored in automatic storage is initialized at the beginning of every call. When a procedure is called recursively, each invocation gets its own copy of the storage.

For a Java method, the STATIC keyword specifies that the method is defined as static. If STATIC is not specified, the method is assumed to be an instance method.

STYLE(style_type)

The STYLE keyword indicates the style type used for the message window (Factor 2 of the DSPLY operation code). The style type can be one of the following figurative constants:
- *INFO
- *WARN
- *HALT

This keyword cannot be used if the MSGDATA, MSGNBR, or MSGTEXT keywords are used.

TIMFMT(format{separator})

The TIMFMT keyword specifies an internal time format, and optionally the time separator, for any of these items of type Time: standalone field; data-structure subfield; prototyped parameter; or return value on a prototype or procedure-interface definition. This keyword is automatically generated for an externally-described data-structure subfield of type Time.

If TIMFMT is not specified, the Time field will have the time format and separator as specified by the TIMFMT keyword on the control specification, if present. If none is specified on the control specification, then it will have *ISO format.

See Table 18 on page 136 for valid formats and separators.

The hierarchy used when determining the internal format and separator for a time array or field is:
1. From the TIMFMT keyword specified on the definition specification
2. From the TIMFMT keyword specified in the control specification
3. *ISO

TOFILE(file_name)

The TOFILE keyword specifies a target file to which a pre-runtime or compile-time array or table is to be written.

If an array or table is to be written, specify the file name of the combined file as the keyword parameter. This file must also be defined in the file description specifications. An array or table can be written to only one output device.

If an array or table is to be written to the same file from which it was read, the same file name that was specified as the FROMFILE parameter must be specified as the TOFILE parameter. This file must be defined as a combined file (C in position 17 on the file-description specification).

VALUE

The VALUE keyword indicates that the parameter is passed by value rather than by reference. Parameters can be passed by value when the procedure they are associated with are called using a procedure call.

When the CLTPGM keyword is used, parameters must be passed by value.

The rules for what can be passed as a value parameter to a called procedure are the same as the rules for what can be assigned using the EVAL operation. The parameter received by the procedure corresponds to the left-hand side of the expression; the passed parameter corresponds to the right-hand side. See “EVAL (Evaluate Expression)” on page 571 for more information.

VARYING

The VARYING keyword indicates that a character, graphic, or UCS-2 field, defined on the definition specifications, should have a variable-length format. If this keyword is not specified for character, graphic or UCS-2 fields, they are defined as fixed length.

For more information, see “Variable-Length Character, Graphic, and UCS-2 Format” on page 113.

Summary According to Definition Specification Type

Table 29 on page 296 lists the required and allowed entries for each definition-specification type.

Table 30 on page 296 and Table 31 on page 297 list the keywords allowed for each definition-specification type.

In each of these tables, an **R** indicates that an entry in these positions is required and an **A** indicates that an entry in these positions is allowed.

Table 29. Required/Allowed Entries for each Definition Specification Type

Type	Pos. 7-21 Name	Pos. 22 External	Pos. 23 DS Type	Pos. 24-25 Defn. Type	Pos. 26-32 From	Pos. 33-39 To / Length	Pos. 40 Data-type	Pos. 41-42 Decimal Pos.	Pos. 44-80 Key-words
Data Structure	A	A	A	R		A			A
Data Structure Subfield	A				A	A	A	A	A
External Subfield	A	R							A
Standalone Field	R			R		A	A	A	A
Named Constant	R			R					R
Prototype	R			R		A	A	A	A
Prototype Parameter	A					A	A	A	A
Procedure Interface	A			R		A	A	A	A
Procedure Interface Parameter	R					A	A	A	A

Table 30. Data Structure, Standalone Fields, Named Constants, and Message Window Keywords

Keyword	Data Structure	Data Structure Subfield	External Subfield	Standalone Field	Named Constant	Message Window
ALIGN	A					
ALT		A	A	A		
ASCEND		A	A	A		
BASED	A			A		
BUTTON						A
CCSID		A		A		
CLASS				A		
CONST (1.)					R	
CTDATA (2.)		A	A	A		
DATFMT		A		A		
DESCEND		A	A	A		
DIM	A	A	A	A		
DTAARA (2.)	A	A		A		
EXTFLD			A			
EXTFMT		A	A	A		
EXTNAME (4.)	A					

Table 30. Data Structure, Standalone Fields, Named Constants, and Message Window Keywords (continued)

Keyword	Data Structure	Data Structure Subfield	External Subfield	Standalone Field	Named Constant	Message Window
FROMFILE (2.)		A	A	A		
INZ	A	A	A	A		
LIKE		A		A		
LIKEDS [5]	A	A				
LIKEREC	A	A				
LINKAGE				R	R	
MSGDATA						A
MSGNBR						A
MSGTEXT						A
MSGTITLE						A
NOOPT	A			A		
OCCURS	A					
OVERLAY		A				
PACKEVEN		A				
PERRCD		A	A	A		
PREFIX (4.)	A					
PROCPTR		A		A		
QUALIFIED	A					
STATIC (3.)	A			A		
STYLE						A
TIMFMT		A		A		
TOFILE (2.)		A	A	A		
VARYING		A		A		

Note:

1 When defining a named constant, the keyword is optional, but the parameter to the keyword is required. For example, to assign a named constant the value '10', you could specify either CONST('10') or '10'.

2 This keyword only applies to global definitions.

3 This keyword only applies to local definitions.

4 This keyword only applies to externally-described data structures.

5 This keyword applies only to program-described data structures.

Table 31. Prototype, Procedure Interface, and Parameter Keywords

Keyword	Prototype (PR)	Procedure Interface (PI)	PR or PI Parameter
ASCEND			A
CCSID	A	A	A
CLASS	A	A	A

Table 31. Prototype, Procedure Interface, and Parameter Keywords (continued)

Keyword	Prototype (PR)	Procedure Interface (PI)	PR or PI Parameter
CLTPGM	A		
CONST			A
DATFMT	A	A	A
DESCEND			A
DIM	A	A	A
DLL	A		
EXTPGM	A		
EXTPROC	A		
LIKE	A	A	A
LIKEDS	A	A	A
LIKEREC	A	A	A
LINKAGE	A		
NOOPT			A
OPTIONS			A
PROCPTR	A	A	A
STATIC	A	A	
TIMFMT	A	A	A
VALUE			A
VARYING	A	A	A

Chapter 19. Input Specifications

For a program described input file, input specifications describe the types of records within the file, the fields within a record, the data within the field, and indicators based on the contents of the fields.

For an externally described file, input specifications are optional and can be used to add functions to the external description.

Detailed information for the input specifications is given in:
- "Program Described Files" on page 300
- "Externally Described Files" on page 307

Input Specification Statement

The general layout for the Input specification is:
- The input specification type (I) is entered in position 6
- The non-commentary part of the specification extends from position 7 to position 80
- The comments section of the specification extends from position 81 to position 100

Program Described

For program described files, entries on input specifications are divided into the following categories:
- Record identification entries (positions 7 through 46), which describe the input record and its relationship to other records in the file.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
IFilename++Sq..RiPos1+NCCPos2+NCCPos3+NCC.................................Comments++++++++++++
I.........And..RiPos1+NCCPos2+NCCPos3+NCC.................................Comments++++++++++++
```

Figure 104. Program Described Record Layout

- Field description entries (positions 31 through 74), which describe the fields in the records. Each field is described on a separate line, below its corresponding record identification entry.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
I.......................Fmt+SPFrom+To+++DcField+++++++++....FrPlMnZr......Comments++++++++++++
```

Figure 105. Program Described Field Layout

Externally Described

For externally described files, entries on input specifications are divided into the following categories:
- Record identification entries (positions 7 through 16, and 21 through 22), which identify the record (the externally described record format) to which VARPG functions are to be added.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
IRcdname+++....Ri.....................................................Comments++++++++++++
```

Figure 106. Externally Described Record Layout

- Field description entries (positions 21 through 30, 49 through 66, and 69 through 74), which describe the VARPG functions to be added to the fields in the record. Field description entries are written on the lines following the corresponding record identification entries.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
I.............Ext-field+.................Field+++++++++......PlMnZr......Comments++++++++++++
```

Figure 107. Externally Described Field Layout

Program Described Files

Program described files include the following entries on input specifications.

Position 6 (Form Type)

An I must appear in position 6 to identify this line as an input specification statement.

Record Identification Entries

Record identification entries (positions 7 through 46) for a program described file describe the input record and its relationship to other records in the file.

Positions 7-16 (File Name)

Entry **Explanation**

A valid file name

This is the same name that appears on the file description specifications for the input file.

Enter the name for the file to be described in these positions. This name must be the same as the name defined for the file on the file description specifications. This file must be an input file, an update file, or a combined file. The file name must be entered on the first record indentification line for each file and can be entered on subsequent record indentification lines for that file. All entries describing one input file must appear together; they cannot be mixed with entries for other files.

Positions 16-18 (Logical Relationship)

Entry **Explanation**

AND More than three identification codes are used.

OR Two or more record types have common fields.

An unlimited number of AND/OR lines can be used. For more information see "AND Relationship" on page 303 and "OR Relationship" on page 303.

Positions 17-18 (Sequence)

Entry **Explanation**

Any two alphanumeric characters
No special sequence checking is done.

Position 19 (Reserved)

Entry **Explanation**

Blank Record types are not checked for a special sequence (positions 17 and 18 have alphanumeric entries).

Position 20 (Option)

Entry **Explanation**

Blank This entry must be blank when positions 17 and 18 contain an alphanumeric entry.

Positions 21-22 (Record Identifying Indicator)

Entry **Explanation**

Blank No indicator is used

01-99 General indicator

LR Control level indicator used for a record identifying indicator

The indicators specified in these positions are used in conjunction with the record identification codes (positions 23 through 46).

Indicators

Positions 21 and 22 associate an indicator with the record type defined on this line. You can enter either 01 to 99 or LR.

When a record is selected for processing and satisfies the conditions indicated by the record identification codes, the appropriate record identifying indicator is set on. This indicator can be used to condition calculation and output operations. Record identifying indicators can be set on or set off by the programmer.

Positions 23-46 (Record Identification Codes)

Entries in positions 23 through 46 identify each record type in the input file. One to three identification codes can be entered on each specification line. More than three record identification codes can be specified on additional lines with the AND/OR relationship. If the file contains only one record type, the identification codes can be left blank; however, a record identifying indicator entry (positions 21 and 22) and a sequence entry (positions 17 and 18) must be made.

Note: Record identification codes are not applicable for graphic UCS-2 data type processing: record identification is done on single byte positions only.

Three sets of entries can be made in positions 23 through 46: 23 through 30, 31 through 38, and 39 through 46. Each set is divided into four groups: , , , . position, not, code part, and character.

The following table shows which categories use which positions in each set.

Category	23-30	31-38	39-46
Position	23-27	31-35	39-43
Not	28	36	44
Code Part	29	37	45
Character	30	38	46

Entries in these sets do not need to be in sequence. For example, an entry can be made in positions 31 through 38 without requiring an entry in positions 23 through 30. Entries for record identification codes are not necessary if input records within a file are of the same type. An input specification containing no record identification code defines the last record type for the file, thus allowing the handling of any record types that are undefined. If no record identification codes are satisfied, control passes to the VARPG exception/error handling routine.

Positions 23-27, 31-35, and 39-43 (Position)

Entry	Explanation
Blank	No record identification code is present.
1-32766	This is the position that contains the record identification code in the record. The position containing the code must be within the record length specified for the file. This entry must be right-adjusted, but leading zeros can be omitted.

Positions 28, 36, and 44 (Not)

Entry	Explanation
Blank	Record identification code must be present.
N	An N in this position means that the record identification code must not be present in the specified record position.

Positions 29, 37, and 45 (Code Part)

This entry specifies what part of the character in the record identification code is to be tested.

Entry	Explanation
C	Entire character
D	Digit

Character (C): The C entry indicates that the complete structure (zone and digit) of the character is to be tested.

Digit (D): The D entry indicates that the digit portion of the character is to be tested. The four right-most bits of the character are compared with the character specified by the position entry.

Positions 30, 38, and 46 (Character)

An entry in this position indicates the identifying character that is compared with the character in the position specified in the input record.

The check for record type starts with the first record type specified. If data in a record satisfies more than one set of record identification codes, the first record type satisfied determines the record types.

When more than one record type is specified for a file, the record identification codes should be coded so that each input record has a unique set of identification codes.

AND Relationship

The AND relationship is used when more than three record identification codes identify a record.

To use the AND relationship, enter at least one record identification code on the first line and enter the remaining record identification codes on the following lines with AND coded in positions 16 through 18 for each additional line used. Positions 7 through 15, 19 through 20, and 46 through 80 of each line with AND in positions 16 through 18 must be blank. Sequence and record-identifying-indicator entries are made in the first line of the group and cannot be specified in the additional lines.

An unlimited number of AND/OR lines can be used on the input specifications.

OR Relationship

The OR relationship is used when two or more record types have common fields.

To use the OR relationship, enter OR in positions 16 and 17. Positions 7 through 15, 18 through 20, and 46 through 80 must be blank. A record identifying indicator can be entered in positions 21 and 22. If the indicator entry is made and the record identification codes on the OR line are satisfied, the indicator specified in positions 21 and 22 on that line is set on. If no indicator entry is made, the indicator on the preceding line is set on.

An unlimited number of AND/OR lines can be used on the input specifications.

Field Description Entries

The field description entries (positions 31 through 74) must follow the record identification entries (positions 7 through 46) for each file.

Position 6 (Form Type)

An I must appear in position 6 to identify this line as an input specification statement.

Positions 7-30 (Reserved)

Positions 7-30 must be blank.

Positions 31-34 (Data Attributes)

Positions 31-34 specify the external format for a date, time, or variable-length character, graphic, or UCS-2 field.

If this entry is blank for a date or time field, then the format/separator specified for the file (with either DATFMT or TIMFMT or both) is used. If there is no external date or time format specified for the file, then an error message is issued. See “Date Data” on page 119 and “Time Data” on page 135 for date and time formats.

The hierarchy used when determining the external date/time format and separator for date and time fields is:

1. The date format and separator specified in positions 31-35
2. From the DATFMT/TIMFMT keyword specified for the current file

3. From the DATFMT/TIMFMT keyword specified in the control specification
4. *ISO

Date and time fields are converted from the external date/time format determined above to the internal format of the date/time field.

For character, graphic, or or UCS-2 data, the *VAR data attribute is used to specify variable-length input fields. If this entry is blank for character, graphic, or UCS-2 data, then the external format must be fixed length. The internal and external format must match, if the field is defined elsewhere in the program. For more information on variable-length fields, see "Variable-Length Character, Graphic, and UCS-2 Format" on page 113.

For more information on external formats, see "Internal and External Formats" on page 103.

Position 35 (Date/Time Separator)

Position 35 specifies a separator character to be used for date/time fields. The & (ampersand) can be used to specify a blank separator. See "Date Data" on page 119 and "Time Data" on page 135 for date and time formats and their default separators.

For an entry to be made in this field, an entry must also be made in positions 31-34 (date/time external format).

Position 36 (Data Format)

The input field is:

Entry	Explanation
Blank	Zoned decimal format or character
A	Character field (fixed- or variable-length format)
N	Character field (Indicator format)
G	Graphic field (fixed- or variable-length format)
C	UCS-2 field (fixed- or variable-length format)
B	Binary format
F	Numeric field (float format)
I	Numeric format (integer format)
L	Numeric field with a preceding (left) plus or minus sign (zoned decimal format)
P	Numeric field (packed decimal format)
R	Numeric field with a following (right) plus or minus sign (zoned decimal format)
S	Numeric field (zoned decimal field)
U	Numeric field (unsigned format)
D	Date field – the date field has the external format specified in positions 31–34 or the default file date format.
T	Time field – the time field has the external format specified in positions 31–34 or the default file time format

Z Timestamp field

The entry in position 36 specifies the data type , and if numeric, the external data format of the data in the program described file. This entry has no effect on the format used for internal processing of the input field in the program.

Positions 37-46 (Field Location)

Entry Explanation

Two 1- to 5-digit numbers
Beginning of a field (from) and end of a field (to).

This entry describes the location and size of each field in the input record. Positions 37 through 41 specify the location of the field's beginning position; positions 42 through 46 specify the location of the field's end position. To define a single-position field, enter the same number in positions 37 through 41 and in positions 42 through 46. Numeric entries must be right-adjusted; leading zeros can be omitted.

The maximum number of positions in the input record for each type of field is:

Number of Positions	Type of Field
31	Zoned decimal numeric (31 digits)
16	Packed numeric (31 digits)
4	Binary (9 digits)
8	Integer (20 digits)
8	Unsigned (20 digits)
8	Float (8 bytes)
32	Numeric with leading or trailing sign (31 digits)
10	Date
8	Time
26	Timestamp
32766	Character (32766 characters)
32766	Variable-Length Character (32764 characters)
32766	Graphic or UCS-2 (16383 double-byte characters)
32766	Variable-Length Graphic or UCS-2 (16382 double-byte characters)
32766	Data structure

The maximum size of a character or data structure field specified as a program described input field is 32766 since that is the maximum record length for a file.

When specifying a variable-length character, graphic, or UCS-2 input field, the length includes the 2 byte length prefix.

For arrays, enter the beginning position of the array in positions 37 through 41 and the ending position in positions 42 through 46. The array length must be an integral multiple of the length of an element. The From-To position does not have to account for all the elements in the array. The placement of data into the array starts with the first element.

Positions 47-48 (Decimal Positions)

Entry Explanation

Blank Character, graphic, UCS-2, float, date, time, or timestamp field.

0-31 Number of decimal positions in numeric field.

This entry, used with the data format entry in position 36, describes the format of the field. An entry in this field identifies the input field as numeric (except float numeric); if the field is numeric, an entry must be made. The number of decimal positions specified for a numeric field cannot exceed the length of the field.

Positions 49-62 (Field Name)

Entry **Explanation**

Symbolic name
Field name, data structure name, data structure subfield name, array name, array element, PAGE, PAGE1-PAGE7, *IN, *INxx, or *IN(xx).

These positions name the fields of an input record that are used in a VARPG program. This name must follow the rules for symbolic names.

To refer to an entire array on the input specifications, enter the array name in positions 49 through 62. If an array name is entered in positions 49 through 62, field indicators (positions 67 through 68) must be blank.

To refer to an element of an array, specify the array name, followed by an index enclosed within parentheses. The index is either a numeric field with zero decimal positions or the actual number of the array element to be used. The value of the index can vary from 1 to n, where n is the number of elements within the array.

Positions 63-64 (Reserved)

Entry **Explanation**

Blank This entry must be blank.

Positions 65-66 (Reserved)

Entry **Explanation**

Blank This entry must be blank.

Positions 67-68 (Field Record Relation)

Entry **Explanation**

Blank The field is common to all record types.

01-99 General indicators.

Field record relation indicators are used to associate fields within a particular record type when that record type is one of several in an OR relationship. This entry reduces the number of lines that must be written.

The field described on a line is extracted from the record only when the indicator coded in positions 67 and 68 is on or when positions 67 and 68 are blank. When positions 67 and 68 are blank, the field is common to all record types defined by the OR relationship.

Positions 69-74 (Field Indicators)

Entry **Explanation**

Blank No indicator specified

01-99 General indicators

Entries in positions 69 through 74 test the status of a field or of an array element as it is read into the program. Field indicators are specified on the same line as the field to be tested. Depending on the status of the field (plus, minus, zero, or blank), the appropriate indicator is set on and can be used to condition later specifications. The same indicator can be specified in two positions, but it should not be used for all three positions. Field indicators cannot be used with arrays that are not indexed.

Positions 69 and 70 (plus) and positions 71 and 72 (minus) are valid for numeric fields only. Positions 73 and 74 can be used to test a numeric field for zeros or a character, graphic, UCS-2 field for blanks.

The field indicators are set on if the field or array element meets the condition specified when the record is read. Each field indicator is related to only one record type; therefore, the indicators are not reset (on or off) until the related record is read again or until the indicator is defined in some other specification.

Externally Described Files

Externally described files include the following entries on input specifications.

Position 6 (Form Type)

An I must appear in position 6 to identify this line as an input specifications statement.

Record Identification Entries

When the description of an externally described file is retrieved, the record definitions are also retrieved. To refer to the record definitions, specify the record format name in the input, calculation, and output specifications of the program. Input specifications for an externally described file are required if:

- Record identifying indicators are to be specified
- A field within a record is to be renamed for the program
- Field indicators are to be used.

The field description specifications must immediately follow the record identification specification for an externally described file.

A record line for an externally described file defines the beginning of the override specifications for the record. All specifications following the record line are part of the record override until another record format name or file name is found in positions 7 through 16 of the input specifications. All record lines that pertain to an externally described file must appear together; they cannot be mixed with entries for other files.

Positions 7-16 (Record Name)

Enter one of the following:

- The external name of the record format. This file name cannot be used for an externally described file.

- The name specified by the RENAME keyword on the file description specifications if the external record format was renamed. A record format name can appear only once in positions 7 through 16 of the input specifications for a program.

Positions 17-20 (Reserved)

Positions 17 through 20 must be blank.

Positions 21-22 (Record Identifying Indicator)

The specification of record identifying indicators in these positions is optional but, if present, follows the rules for . Program Described Files. See "Program Described Files" on page 300.

Positions 23-80 (Reserved)

Positions 23-80 must be blank.

Field Description Entries

The field description specifications for an externally described file can be used to rename a field within a record for a program or to specify field indicator functions. The field definitions (attributes) are retrieved from the externally described file and cannot be changed by the program. If the attributes of a field are not valid for a VARPG program the field cannot be used. Diagnostic checking is done on fields contained in an external record format in the same way as for source statements.

Normally, externally described input fields are only read during input operations if the field is actually used elsewhere in the program.

Positions 7-20 (Reserved)

Positions 7 through 20 must be blank.

Positions 21-30 (External Field Name)

If a field within a record in an externally described file is to be renamed, enter the external name of the field in these positions. A field may have to be renamed because the name is the same as a field name specified in the program and two different names are required.

Note: If the input field is for a file that has the PREFIX keyword coded, and the prefixed name has already been specified in the Field Name entry (positions 49 - 62) of a prior Input specification for the same record, then the prefixed name must be used as the external name. For more information, see "PREFIX(prefix{:nbr_of_char_replaced})" on page 249.

Positions 31-48 (Reserved)

Positions 31 through 48 must be blank.

Positions 49-62 (Field Name)

The field name entry is made only when it is required for the functions such as control levels added to the external description. The field name entry contains one of the following:

- The name of the field as defined in the external record description (if 10 characters or less)

- The name specified to be used in the program that replaced the external name specified in positions 21 through 30.

The field name must follow the rules for using symbolic names.

Indicators cannot be null-capable.

Positions 63-64 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Positions 65-66 (Reserved)

Entry	Explanation
Blank	This entry must be blank.

Positions 67-68 (Reserved)

Positions 67 and 68 must be blank.

Positions 69-74 (Field Indicators)

Entry	Explanation
Blank	No indicator specified
01-99	General indicators

Field indicators are allowed for null-capable fields only if the **User control** option or **ALWNULL(*USRCTL)** keyword is specified.

If a null-capable field contains a null value, the indicator is set off.

See "Positions 69-74 (Field Indicators)" on page 306 for program described files.

Positions 75-80 (Reserved)

Positions 75 through 80 must be blank.

Chapter 20. Calculation Specifications

Calculation specifications indicate the operations done on the data in a program.

You can specify calculation specifications in three different formats:
- "Traditional Syntax"
- "Extended Factor 2 Syntax" on page 316
- "Free-Form Syntax" on page 318.

See Chapter 26, "Operation Code Details," on page 501 for details on how the calculation specification entries must be specified for individual operation codes.

Traditional Syntax

The general layout for the calculation specification is as follows:
- The calculation specification type (C) is entered in position 6
- The non-comment part of the specification extends from position 7 to position 80. These positions are divided into three parts that specify the following:
  - When calculations are done: The conditioning indicators specified in positions 7 through 11 determine when and under what conditions the calculations are to be done.
  - What kind of calculations are done: The entries specified in positions 12 through 70 (12 through 80 for operations that use extended-factor 2, see "Extended Factor 2 Syntax" on page 316 and Chapter 24, "Expressions," on page 381) specify the kind of calculations done, the data (such as fields or files) upon which the operation is done, and the field that contains the results of the calculation.
  - What tests are done on the results of the operation: Indicators specified in positions 71 through 76 are used to test the results of the calculations and can condition subsequent calculations or output operations. The resulting indicator positions have various uses, depending on the operation code.
- The comments section of the specification extends from position 81 to position 100

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments++++++++++++
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 108. Calculation Specification Layout

Calculation-Specification Extended-Factor 2 Continuation Line

The Extended-Factor 2 field can be continued on subsequent lines as follows:
- Position 6 of the continuation line must contain a C
- Positions 7 to 35 of the continuation line must be blank
- The specification continues on or past position 36

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
C.............................Extended-factor2-continuation++++++++++++++++Comments++++++++++++
```

Figure 109. Calculation-Specification Extended-Factor 2 Continuation Line

Position 6 (Form Type)

A C must appear in position 6 to identify this line as a calculation specification statement.

Positions 7-8 (Control Level)

Entry	Explanation
Blank	The calculation operation is done if the indicators in positions 9 through 11 allow it; or the calculation is part of a subroutine. Blank is also used for declarative operation codes.
SR	The calculation operation is part of a subroutine. A blank entry is also valid for calculations that are part of a subroutine.
AN, OR	Indicators on more than one line condition the calculation.

Subroutine Identifier

An SR entry in positions 7 and 8 may optionally be used for operations within subroutines as a documentation aid. The operation codes BEGACT and ENDACT serve as delimiters for an action subroutine. The operation codes BEGSR and ENDSR serve as delimiters for a subroutine.

AND/OR Lines Identifier

Positions 7 and 8 can contain AN or OR to define additional indicators (positions 9 through 11) for a calculation.

The entry in positions 7 and 8 of the line immediately preceding an AND/OR line or a group of AND/OR lines determines when the calculation is to be processed. The entry in positions 7 and 8 on the first line of a group applies to all AND/OR lines in the group.

Positions 9-11 (Indicators)

Positions 10 and 11 contain an indicator that is tested to determine if a particular calculation is to be processed:

Entry	Explanation
Blank	The operation is processed on every record
01-99	General indicators
LR	Last record indicator

A blank in position 9 designates that the indicator must be on for a calculation to be done. An N in positions 9 designates that the associated indicator must be off for a calculation to be done.

Positions 12-25 (Factor 1)

Factor 1 names a field or gives actual data (literals) on which an operation is done, or contains a VARPG special word (for example, *LOCK) which provides extra

information on how an operation is to be done. The entry must begin in position 12. The entries for factor 1 depend on the operation code specified in positions 26 through 35. For the specific entries for factor 1 for a particular operation code, see Chapter 26, "Operation Code Details," on page 501. With some operation codes, two operands may be specified separated by a colon.

Positions 26-35 (Operation and Extender)

Positions 26 through 35 specify the kind of operation to be done using factor 1, factor 2, and the result field entries. The operation code must begin in position 26. For further information on the operation codes, see Chapter 23, "Operations," on page 341 and Chapter 26, "Operation Code Details," on page 501.

Operation Extender

Entry	Explanation
Blank	No operation extension supplied
H	Half adjust (round) result of numeric operation and set resulting indicators according to the value of the result field after half-adjusting has been done
N	Record is read but not locked for READ, READE, READP, READPE, or CHAIN operations on an update disk file
	Set pointer to *NULL after successful DEALLOC
P	Pad the result field with blanks if the result field is longer than the result of the operation
	Pass operational descriptors on a bound call
D	Date field
T	Time field
Z	Timestamp field
M	Default precision rules
R	″Result Decimal Position″ precision rules
E	Error handling

The operation extenders provide additional attributes to the operations that they accompany. Operation extenders are specified in positions 26-35 of calculation specifications. They must begin to the right of the operation code and be contained within parentheses; blanks can be used for readability. For example, the following are valid entries: MULT(H), MULT (H), MULT (H).

More than one operation extender can be specified. For example, the EVAL operation can specify both half-adjust and the default precision rules with EVAL(HM).

An H indicates whether the contents of the result field are to be half adjusted (rounded). Resulting indicators are set according to the value of the result field after half-adjusting has been done.

An N in a READ, READE, READP, READPE, or CHAIN operation on an update disk file indicates that a record is to be read, but not locked. If no value is specified, the default action of locking occurs.

An N in a DEALLOC operation indicates that the result field pointer is to be set to *NULL after a successful deallocation.

A P indicates that the result field is padded after executing the instruction if the result field is longer than the result of the operation.

The D, T, and Z extenders can be used with the TEST operation code to indicate a date, time, or timestamp field.

M and R are specified for the precision of single free-form expressions. For more information, see "Precision Rules for Numeric Operations" on page 390.

M indicates that the default precision rules are used.

R indicates that the precision of a decimal intermediate will be computed such that the number of decimal places will never be reduced smaller than the number of decimal positions of the result of the assignment.

E indicates that operation-related errors will be checked with built-in function %ERROR.

Positions 36-49 (Factor 2)

Factor 2 names a field, record format or file, or gives actual data on which an operation is to be done, or contains a special word (for example, *ALL) which gives extra information about the operation to be done. The entry must begin in position 36. The entries that are valid for factor 2 depend on the operation code specified in positions 26 through 35. With some operation codes, two operands may be specified separated by a colon. For the specific entries for factor 2 for a particular operation code, see Chapter 26, "Operation Code Details," on page 501.

Positions 50-63 (Result Field)

The result field names the field or record format that contains the result of the calculation operation specified in positions 26 through 35. The field specified must be modifiable. For example, it cannot be a user date field. With some operation codes, two operands may be specified separated by a colon. See Chapter 26, "Operation Code Details," on page 501 for the result field rules for individual operation codes.

Positions 64-68 (Field Length)

Entry	Explanation
1-31	Numeric field length.
1-65535	Character field length.
Blank	The result field is defined elsewhere or a field cannot be defined using this operation code

Positions 64 through 68 specify the length of the result field. This entry is optional, but can be used to define a numeric or character field not defined elsewhere in the program. These definitions of the field entries are allowed if the result field contains a field name. Other data types must be defined on the definition specification or on the calculation specification using the *LIKE DEFINE operation.

The entry specifies the number of positions to be reserved for the result field. The entry must be right-adjusted. The unpacked length (number of digits) must be specified for numeric fields.

If the result field is defined elsewhere in the program, no entry is required for the length. However, if the length is specified, and if the result field is defined elsewhere, the length must be the same as the previously defined length. If the result field length is different from the previously defined length, the previously defined value is used.

Positions 69-70 (Decimal Positions)

Entry	Explanation
Blank	The result field is character data, has been defined elsewhere in the program, or no field length has been specified.
0-31	Number of decimal positions in a numeric result field: • If the numeric result field contains no decimal positions, enter a '0' (zero). • The number of decimal positions specified cannot exceed the length of the field.

Positions 69-70 indicate the number of positions to the right of the decimal in a numeric result field.

Positions 71-76 (Resulting Indicators)

These positions can be used to test the value of a result field after the completion of an operation, or to indicate conditions like end-of-file, error, or record-not-found. For some operations, you can control the way the operation is performed by specifying different combinations of the three resulting indicators (for example, LOOKUP). The resulting indicator positions have different uses, depending on the operation code specified. See the individual operation codes in Chapter 26, "Operation Code Details," on page 501 for a description of the associated resulting indicators. For arithmetic operations, the result field is tested only after the field is truncated and half-adjustment is done (if specified). The setting of indicators depends on the results of the tests specified.

Entry	Explanation
Blank	No resulting indicator specified
01-99	General indicators
LR	Last record indicator

Resulting indicators cannot be used when the result field uses a non-indexed array.

If the same indicator is used as a resulting indicator on more than one calculation specification, the most recent specification processed determines the status of that indicator.

Note: When the calculation operation is done, the specified resulting indicators are set off, and, if a condition specified by a resulting indicator is satisfied, that indicator is set on.

Extended Factor 2 Syntax

Certain operation codes allow an expression to be used in the extended-factor 2 field.

Positions 7-8 (Control Level)

See "Positions 7-8 (Control Level)" on page 312.

Positions 9-11 (Indicators)

See "Positions 9-11 (Indicators)" on page 312.

Positions 12-25 (Factor 1)

Factor 1 must be blank.

Positions 26-35 (Operation and Extender)

Positions 26 through 35 specify the kind of operation to be done using the expression in the extended-factor 2 field. The operation code must begin in position 26. For further information on the operation codes, see Chapter 23, “Operations,” on page 341 and Chapter 26, “Operation Code Details,” on page 501.

The program processes the operations in the order specified on the calculation specifications form.

Operation Extender

Entry	Explanation
Blank	No operation extension supplied.
H	Half adjust (round) result of numeric operation
M	Default precision rules
R	″Result Decimal Position″ precision rules
E	Error handling

Half adjust may be specified, using the H extender, on arithmetic EVAL and RETURN operations.

The type of precision may be specified, using the M or R extender, on DOU, DOW, EVAL, IF, RETURN, and WHEN operations.

Error handling may be specified, using the ’E’ extender.

Positions 36-80 (Extended-Factor 2)

A free form syntax is used in this field. It consists of combinations of operands and operators, and may optionally span multiple lines. If specified across multiple lines, the continuation lines must be blank in positions 7-35

The operations that take an extended factor 2 are:
- "CALLP (Call a Prototyped Procedure or Program)" on page 522
- "DOU (Do Until)" on page 556
- "DOW (Do While)" on page 559
- "EVAL (Evaluate Expression)" on page 571
- "EVALR (Evaluate expression, right adjust)" on page 573
- "FOR (For)" on page 581
- "IF (If)" on page 586
- "ON-ERROR (On Error)" on page 641
- "RETURN (Return to Caller)" on page 671
- "WHEN (When True Then Select)" on page 713

See the specific operation codes for more information. See "Continuation Rules" on page 215 for more information on coding continuation lines.

Free-Form Syntax

To begin a free-form calculation group, specify `/FREE` in positions 7 to 11 and leave positions 12 to 80 blank. The free-form calculation block ends when you specify `/END-FREE`.

In a free-form statement, the operation code does not need to begin in any specific position within columns 8–80. Any extenders must appear immediately after the operation code on the same line, within parentheses. There must be no embedded blanks between the operation code and extenders. Following the operation code and extenders, you specify the Factor 1, Factor 2, and the Result Field operands separated by blanks. If any of these are not required by the operation, you may leave them out. You can freely use blanks and continuation lines in the remainder of the statement. Each statement must end with a semicolon. The remainder of the record after the semicolon must be blank or contain an end-of-line comment.

For the EVAL or CALLP operation code, you can omit the operation code. For example, the following two statements are equivalent:

```
eval pos = %scan (',': name);
pos = %scan (',': name);
```

For each record within a free-form calculation block, positions 6 and 7 must be blank.

You can specify compiler directives within a free-format calculation block, with the following restrictions:
- The compiler directive must be the first item on the line. Code the directive starting anywhere from column 7 onward. It cannot continue to the next line.
- Compiler directives are not allowed within a statement. The directive must appear on a new line after one statement ends and before the next statement begins.
- Any statements that are included by a /COPY or /INCLUDE directive are considered fixed syntax calculations. Any free-form statements in a /COPY member must be delimited by the /FREE and /END-FREE directives.

Free-form operands can be longer than 14 characters. The following are not supported:

- Continuation of numeric literals
- Defining field names
- Resulting indicators. (In most cases where you need to use operation codes with resulting indicators, you can use an equivalent built-in function instead.)

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/free

        read  file;              // Get next record
        dow not %eof(file);      // Keep looping while we have
                                 // a record
            if %error;
                dsply 'The read failed';
                leave;
            else;
                chain(n) name database data;
                time = hours * num_employees
                         + overtime_saved;
                pos = %scan (',': name);
                name = %xlate(upper:lower:name);
                exsr handle_record;
                read file;
            endif;
        enddo;

     begsr handle_record;
        eval(h) time = time + total_hours_array (empno);
        temp_hours = total_hours - excess_hours;
        record_transaction();
     endsr;

/end-free
```

Figure 110. Example of Free-Form Calculation Specification

You can combine free-form and traditional calculation specifications in the same program, as shown below:

```
C                   testb     OPEN_ALL      flags                   10
 /free
          if *in10;
            openAllFiles();
          endif;
 /end-free
```

Figure 111. Example that Combines Traditional and Free-Form Calculation Specifications

Positions 8-80 (Free-form Operations)

Enter an operation that is supported in free-form syntax. Code an operation code (EVAL and CALLP are optional) followed by the operands or expressions. The operation may optionally span multiple lines. No new continuation characters are required; each statement ends with a semicolon (;). However, existing continuation rules still apply.

See "Continuation Rules" on page 215 for more information on coding continuation lines.

The operation codes that can use free-form syntax are listed below. For operations that cannot use free-form syntax, check the detailed description in Chapter 26, "Operation Code Details," on page 501 to see if there is a suggested replacement.

- "BEGSR (Begin User Subroutine)" on page 511
- "CALLP (Call a Prototyped Procedure or Program)" on page 522
- "CHAIN (Random Retrieval from a File)" on page 529
- "CLEAR (Clear)" on page 539
- "CLOSE (Close Files)" on page 542
- "COMMIT (Commit)" on page 544
- "DEALLOC (Free Storage)" on page 546
- "DELETE (Delete Record)" on page 551
- "DOU (Do Until)" on page 556
- "DOW (Do While)" on page 559
- "DSPLY (Display Message Window)" on page 562
- "ELSE (Else)" on page 564
- "ELSEIF (Else If)" on page 565
- "ENDyy (End a Structured Group)" on page 566
- "ENDSR (End of User Subroutine)" on page 569
- "EVAL (Evaluate Expression)" on page 571
- "EVALR (Evaluate expression, right adjust)" on page 573
- "EXCEPT (Calculation Time Output)" on page 575
- "EXSR (Invoke User Subroutine)" on page 577
- "FEOD (Force End of Data)" on page 580
- "FOR (For)" on page 581
- "IF (If)" on page 586
- "IN (Retrieve a Data Area)" on page 589
- "ITER (Iterate)" on page 591
- "LEAVE (Leave a Do/For Group)" on page 596
- "LEAVESR (Leave a Subroutine)" on page 598
- "MONITOR (Begin a Monitor Group)" on page 602
- "ON-ERROR (On Error)" on page 641
- "OPEN (Open File for Processing)" on page 642
- "OTHER (Otherwise Select)" on page 645
- "OUT (Write a Data Area)" on page 646
- "POST (Post)" on page 652
- "READ (Read a Record)" on page 653
- "READC (Read Next Changed Record)" on page 656
- "READE (Read Equal Key)" on page 658
- "READP (Read Prior Record)" on page 661
- "READPE (Read Prior Equal)" on page 663
- "RESET (Reset)" on page 668
- "RETURN (Return to Caller)" on page 671
- "ROLBK (Roll Back)" on page 672
- "SELECT (Begin a Select Group)" on page 676
- "SETGT (Set Greater Than)" on page 679
- "SETLL (Set Lower Limit)" on page 681
- "SORTA (Sort an Array)" on page 686
- "TEST (Test Date/Time/Timestamp)" on page 700
- "UNLOCK (Unlock a Data Area or Release a Record)" on page 709
- "UPDATE (Modify Existing Record)" on page 711
- "WHEN (When True Then Select)" on page 713
- "WRITE (Create New Records)" on page 717

Chapter 21. Output Specifications

Output specifications describe the record and the format of fields in a program described output file and when the record is to be written. Output specifications are optional for an externally described file. If NOMAIN is coded on a control specification, only exception output can be done.

Output specifications can be divided into two categories: record identification and control (positions 7 through 51), and field description and control (positions 21 through 80). These specifications are entered on the Output Specifications.

Detailed information for the output specifications is given in:
- "Program Described Files" on page 322.
- "Externally Described Files" on page 332.

The following rules apply for output files:
- DISK files:
  - DISK files can be either remote or local
  - Remote files must be externally described
  - Local files must be program described
- PRINTER files:
  - PRINTER files must be program described
- SPECIAL files:
  - SPECIAL files must be program described.

Output Specification Statement

The general layout for the Output specification is:
- The output specification type (O) is entered in position 6
- The non-comment part of the specification extends from position 7 to position 80
- The comments section of the specification extends from position 81 to position 100

Program Described

For program described files, entries on the output specifications can be divided into two categories:
- Record identification and control (positions 7 through 51)

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
OFilename++EF..N01N02N03Excnam++++B++A++Sb+Sa+.........................Comment++++++++++++
OFilename++EAddN01N02N03Excnam++++.....................................Comment++++++++++++
O.........And..N01N02N03Excnam++++.....................................Comment++++++++++++
```

Figure 112. Program Described Record Layout

- Field description and control (positions 21 through 80). Each field is described on a separate line, below its corresponding record identification entry.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
O.............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++Comment++++++++++++
O.........................................Constant/editword-ContinutioComment++++++++++++
```

Figure 113. Program Described Field Layout

Externally Described

For externally described files, entries on output specifications are divided into the following categories:

- Record identification and control (positions 7 through 39)

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
ORcdname+++E...N01N02N03Excnam++++.....................................Comment++++++++++++
ORcdname+++EAddN01N02N03Excnam++++.....................................Comment++++++++++++
O.........And..N01N02N03Excnam++++.....................................Comment++++++++++++
```

Figure 114. Externally Described Record Layout

- Field description and control (positions 21 through 43, and 45).

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
O.............N01N02N03Field+++++++++.B.................................Comment++++++++++++
```

Figure 115. Externally Described Field Layout

Program Described Files

Program described files include the following entries on output specifications.

Position 6 (Form Type)

An O must appear in position 6 to identify this line as an output specifications statement.

Record Identification and Control Entries

Entries in positions 7 through 51 identify the output records that make up the files, provide the correct spacing on printed reports, and determine under what conditions the records are to be written.

Positions 7-16 (File Name)

Entry Explanation

A file name

Specify the file name on the first line that defines an output record for the file. The file name specified must be the same file name assigned to the output, update, or combined file on the file description specifications. If records from files are interspersed on the output specifications, the file name must be specified each time the file changes.

For files specified as output, update, combined, or input with ADD, at least one output specification is required unless an explicit file operation code with a data

structure name specified in the result field is used in the calculations. For example, a WRITE operation does not require output specifications.

Positions 16-18 (Logical Relationship)

Entry	Explanation
AND or OR	AND/OR indicates a relationship between lines of output indicators. AND/OR lines are valid for output records, but not for fields. To specify this relationship, enter AND/OR in positions 16 through 18 on each additional line following the line containing the file name. At least one indicator must be specified on each AND line. An unlimited number of AND/OR lines can be specified on the output specifications.

Positions 7 through 15 must be blank when AND/OR is specified.

Position 17 (Type - Program Described File)

Entry	Explanation
E	Exception records are written during calculation processing. Exception records can be specified only when the operation code EXCEPT is used. See Chapter 23, "Operations," on page 341 for more information on the EXCEPT operation code.

Positions 18-20 (Record Addition/Deletion)

Entry	Explanation
ADD	Add a record to the input file, output file, update file, or subfile. For local files, all records are added to the end of the file. Updates take place at the current record.
DEL	Delete the last record read from the file. The deleted record cannot be retrieved; the record is deleted from the system.

Positions 21-29 (File Record ID Indicators)

Entry	Explanation
Blank	The line or field is output every time the record is checked for output.
01-99	A general indicator that is used as a resulting indicator, field indicator, or record identifying indicator.
LR	Last record indicator.

Conditioning indicators are not required on output lines. If conditioning indicators are not specified, the line is output every time that record is checked for output. Up to three indicators can be entered on one specification line to control when a record or a particular field within a record is written. The indicators that condition the output are coded in positions 22 and 23, 25 and 26, and 28 and 29. When an N is entered in positions 21, 24, or 27, the indicator in the associated position must be off for the line or field to be written. Otherwise, the indicator must be on for the line or field to be written. See "PAGE, PAGE1-PAGE7" on page 327 for information on how output indicators affect the PAGE fields.

If more than one indicator is specified on one line, all indicators are considered to be in an AND relationship.

If the output record must be conditioned by more than three indicators in an AND relationship, enter the letters AND in positions 16 through 18 of the following line and specify the additional indicators in positions 21 through 29 on that line.

Positions 40 through 51 (spacing and skipping) must be blank for all AND lines.

If the output record is to be written when any one of two or more sets of conditions exist (an OR relationship), enter the letters OR in positions 16-18 of the following specification line, and specify the additional OR indicators on that line.

When an OR line is specified for a printer file, the skip and space entries (positions 40 through 51) can all be blank, in which case the space and skip entries of the preceding line are used. If they differ from the preceding line, enter space and skip entries on the OR line.

Positions 30-39 (EXCEPT Name)

When the record type is an exception record (indicated by an E in position 17), a name can be placed in these positions of the record line. The EXCEPT operation can specify the name assigned to a group of the records to be output. This name is called an EXCEPT name. An EXCEPT name must follow the rules for using symbolic names. A group of any number of output records can use the same EXCEPT name, and the records do not have to be consecutive records.

When the EXCEPT operation is specified without an EXCEPT name, only those exception records without an EXCEPT name are checked and written if the conditioning indicators are satisfied.

When the EXCEPT operation specifies an EXCEPT name, only the exception records with that name are checked and written if the conditioning indicators are satisfied.

The EXCEPT name is specified on the main record line and applies to all AND/OR lines.

An EXCEPT operation with no fields can be used to release a record lock in a file. The UNLOCK operation can also be used for this purpose. In the following figure, the record lock in file RCDA is released by the EXCEPT operation.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C     KEY           CHAIN     RCDA
C                   EXCEPT    RELESE
ORcdname+++D...N01N02N03Excnam++++.....................................
O
O*
ORCDA      E            RELESE
O*                    (no fields)
```

Figure 116. Record Lock in File Released by EXCEPT Operation

Positions 40-51 (Space and Skip)

Use positions 40 through 51 to specify line spacing and skipping for a printer file. Spacing refers to advancing one line at a time, and skipping refers to jumping from one print line to another.

If spacing and skipping are specified for the same line, the spacing and skipping operations are processed in the following sequence:
1. Skip before
2. Space before
3. Print a line
4. Skip after
5. Space after.

If the PRTCTL (printer control option) keyword is not specified on the file description specifications, an entry must be made in one of the following positions when the device is PRINTER: 40-42 (space before), 43-45 (space after), 46-48 (skip before), or 49-51 (skip after). If a space/skip entry is left blank, the particular function with the blank entry (such as space before or space after) does not occur. If entries are made in positions 40-42 (space before) or in positions 46-51 (skip before and skip after) and no entry is made in positions 43 - 45 (space after), no space occurs after printing. When PRTCTL is specified, it is used only on records with blanks specified in positions 40 through 51.

If a skip before or a skip after a line on a new page is specified, but the printer is on that line, the skip does not occur.

Positions 40-42 (Space Before)

Entry	Explanation
0 or Blank	No spacing
1-255	Spacing values

Positions 43-45 (Space After)

Entry	Explanation
0 or Blank	No spacing
1-255	Spacing values

Positions 46-48 (Skip Before)

Entry	Explanation
Blank	No skipping occurs.
1-255	Skipping values

Positions 49-51 (Skip After)

Entry	Explanation
Blank	No skipping occurs.
1-255	Skipping values

Field Description and Control Entries

Each field is described on a separate line. These entries determine under what conditions and in what format fields of a record are to be written. Field description and control information for a field begins on the line following the record identification line.

Positions 21-29 (Output Indicators)

Indicators specified on the field description lines determine whether a field is to be included in the output record, except for PAGE reserved fields. See "PAGE, PAGE1-PAGE7" on page 327 for information on how output indicators affect the PAGE fields. The same types of indicators can be used to control fields as are used to control records, see "Positions 21-29 (File Record ID Indicators)" on page 324. Indicators used to condition field descriptions lines cannot be specified in an AND/OR relationship.

Positions 30-43 (Field Name)

In positions 30 through 43, use one of the following entries to specify each field that is to be written out:

- A field name
- Blanks if a constant is specified in positions 53 through 80
- A table name, array name, or array element
- A named constant
- The reserved words PAGE, PAGE1 through PAGE7, *PLACE, UDATE, *DATE, UDAY, *DAY, UMONTH, *MONTH, UYEAR, *YEAR, *IN, *INxx, or *IN(xx)
- A data structure name or data structure subfield name.

Note: A pointer field is not a valid output field, that is, pointer fields cannot be written.

Field Names, Blanks, Tables, and Arrays

The field names used must be defined in the program. Do not enter a field name if a constant is used in positions 53-80. If a field name is entered in positions 30 through 43, positions 7 through 20 must be blank.

Fields can be specified in any order because the sequence in which they appear on the output records is determined by the entry in positions 47 through 51. If fields overlap, the last field specified is the only field completely written.

When a non-indexed array name is specified, the entire array is written. An array name with a constant index or variable index causes one element to be written. When a table name is specified, the element last found in a LOOKUP operation is written. The first element of a table is written if no successful LOOKUP operation was done.

The conditions for a record and the field it contains must be satisfied before the field is written out.

PAGE, PAGE1-PAGE7

To use automatic page numbering, enter PAGE in positions 30 through 43 as the name of the output field. Indicators specified in positions 21 though 29 condition the resetting of the PAGE field, not whether it prints. The PAGE field is always incremented by 1 and printed. If the conditioning indicators are met, it is reset to zero before being incremented by 1 and printed. If page numbers are needed for several output files (or for different numbering within one file), the entries PAGE1 through PAGE7 can be used. The PAGE fields are automatically zero-suppressed by the Z edit code.

For more information on the PAGE reserved words, see "Words with Special Functions and Reserved Words" on page 5.

*PLACE

*PLACE is used to repeat data in an output record. Fields or constants that have been specified on previous specification lines can be repeated in the output record without having the field and end positions named on a new specification line. When *PLACE is entered in positions 30 through 43, all data between the first position and the highest end position previously specified for a field in that output record is repeated until the end position specified in the output record on the *PLACE specification line is reached. The end position specified on the *PLACE specification line must be at least twice the highest end position of the group of fields to be duplicated. *PLACE can be used with any type of output. Blank after (position 45), editing (positions 44, 53 through 80), data format (position 52), and relative end positions cannot be used with *PLACE.

User Date Reserved Words

UDATE, *DATE, UDAY, *DAY, UMONTH, *MONTH, UYEAR, and *YEAR allow you to supply a date for the program at run time. For more information on the user date words, see "User Date Special Words" on page 8.

*IN, *INxx, *IN(xx)

*IN, *INxx and *IN(xx) allow you to refer to and manipulate indicators as data.

Position 44 (Edit Codes)

Entry	Explanation
Blank	No edit code is used.
1-4, A-D, J-Q, X, Y, Z	Numeric fields are zero-suppressed and punctuated according to a predefined pattern without the use of edit words.

Position 44 is used to specify edit codes that suppress leading zeros in a numeric field or to punctuate a numeric field without using an edit word. Allowable entries are 1 through 4, A through D, J through Q, X, Y, Z, and blank.

Position 45 (Blank After)

Entry	Explanation
Blank	The field is not reset. This position must be blank for look-ahead, user date reserved words, *PLACE, named constants, and literals.
B	The field specified in positions 30 through 43 is reset to blank, zero, or the default date/time/timestamp value after the output operation is complete.

Position 45 is used to reset a numeric field to zeros or a character, graphic, or UCS-2 field to blanks. Date, time, and timestamp fields are reset to their default values.

If the field is conditioned by indicators in positions 21 through 29, the blank after is also conditioned.

If blank after (position 45) is specified for a field to be written more than once, the B should be entered on the last line specifying output for that field, or else the field named will be printed as the blank-after value for all lines after the one doing the blank after.

Positions 47-51 (End Position)

Entry	Explanation
1-n	End position

Positions 47 through 51 define the end position of a field or constant on the output record.

Valid entries for end positions are blanks, +nnnn, –nnnn, and nnnnn. All entries in these positions must end in position 51. Enter the position of the rightmost character of the field or constant. The end position must not exceed the record length for the file.

If an entire array is to be written, enter the end position of the last element in the array in positions 47 through 51. If the array is to be edited, be careful when specifying the end position to allow enough positions to write all edited elements. Each element is edited according to the edit code or edit word.

The +nnnn or –nnnn entry specifies the placement of the field or constant relative to the end position of the previous field. The number (nnnn) must be right adjusted, but leading zeros are not required. Enter the sign anywhere to the left of the number within the entry field. To calculate the end position, use these formulas:

```
end position = previous end position +nnnn + field length
end position = previous end position -nnnn + field length
```

For the first field specification in the record, the previous end position is equal to zero. The field length is the length of the field after editing, or the length of the constant specified in this specification. The use of +nnnn is equivalent to placing nnnn positions between the fields. A -nnnn causes an overlap of the fields by nnnn positions. For example, if the previous end position is 6, the number of positions to be placed between the fields (nnnn) is 5, and the field length is 10, the end position equals 21.

When *PLACE is used, an actual end position must be specified; it cannot be blank or a displacement.

An entry of blank is treated as an entry of +0000. No positions separate the fields.

Position 52 (Data Format)

Entry	Explanation
Blank	This position must be blank if editing is specified. • For numeric fields the data is written in zoned decimal format. • For float numeric fields, the data is to be written in the external display representation. • For UCS-2 fields, the data is to be written in UCS-2 format. • For date, time, and timestamp fields the data is written without format conversion performed. • For character fields the data is to be written as it is stored.
A	Valid for Character fields only. The character field is to be written in either fixed- or variable-length format depending on the absense or presence of the *VAR data attribute.
N	The character field is to be written in indicator format.
C	The UCS-2 field is to be written in either fixed- or variable-length format depending on the absense or presence of the *VAR data attribute.
G	Valid for Graphic fields in program-described files only. The graphic field will be written in either fixed- or variable-length format depending on the absense or presence of the *VAR data attribute.
B	The numeric field is to be written in binary format.
F	The numeric field is to be written in float format
I	The numeric field is to be written out in integer format.
L	The numeric field is written with a preceding (left) plus or minus sign, in zoned-decimal format.
P	The numeric field is to be written in packed-decimal format.
R	The numeric field is written with a following (right) plus or minus sign, in zoned-decimal format.
S	The numeric field is to be written out in zoned decimal format.
U	The numeric field is to be written out in unsigned integer format.
D	Date field– the date field is converted to the format specified in positions 53-80 or to the default file date format.
T	Time field– the time field is converted to the format specified in positions 53-80 or to the default file time format.
Z	Valid for Timestamp fields only.

The entry in position 52 specifies the external format of the data in the records in the file. This entry has no effect on the format used for internal processing of the output field in the program.

For numeric fields, the number of bytes required in the output record depends on this format. For example, a numeric field with 5 digits requires:
- 5 bytes when written in zoned format
- 3 bytes when written in packed format
- 6 bytes when written in either L or R format
- 4 bytes when written in binary format

- 2 bytes when written in either I or U format. This may cause an error at run time if the value is larger than the maximum value for a 2–byte integer or unsigned field. For the case of 5–digit fields, binary format may be better.

  Float numeric fields written out with blank Data Format entry occupy either 14 or 23 positions (for 4–byte and 8–byte float fields respectively) in the output record.

Note: A 'G' or blank must be specified for a graphic field in a program-described file.

Positions 53-80 (Constant, Edit Word, Data Attribute)

Positions 53 through 80 are used to specify a constant, an edit word, or a data attribute.

Constants

Constants consist of character data (literals) that does not change from one processing of the program to the next. A constant is the actual data used in the output record rather than a name representing the location of the data.

A constant can be placed in positions 53 through 80. The constant must begin in position 54 (apostrophe in position 53), and it must end with an apostrophe even if it contains only numeric characters. Any apostrophe used within the constant must be entered twice; however, only one apostrophe appears when the constant is written out. The field name (positions 30 through 43) must be blank. Constants can be continued. (See "Continuation Rules" on page 215 for continuation rules.) Instead of entering a constant, you can use a named constant.

Graphic and UCS-2 literals or named constants are not allowed as edit words, but may be specified as constants.

Edit Word

An edit word specifies the punctuation of numeric fields, including the printing of dollar signs, commas, periods, and sign status.See "Parts of an Edit Word" on page 199 for details.

Edit words must be character literals or named constants. Graphic, UCS-2 or hexadecimal literals or named constants are not allowed.

Data Attributes

Data attributes specify the external format for a date, time, or variable-length character, graphic, or UCS-2 field.

For date and time data, if no date or time format is specified, then the format/separator specified for the file (with either DATFMT or TIMFMT or both) is used. If there is no external date or time format specified for the file, then an error message is issued. See "DATFMT(fmt{separator})" on page 227 and "TIMFMT(fmt{separator})" on page 235 for date and time formats.

The hierarchy used when determining the external date/time format and separator for date and time fields is:

1. The date format and separator specified in positions 53-58 (or 53-57).
2. From the DATFMT/TIMFMT keyword specified for the current file
3. From the DATFMT/TIMFMT keyword specified in the control specification
4. *ISO

Date and time fields are converted from their internal date/time format to the external format determined above.

For character, graphic, and UCS-2 data, the *VAR data attribute is used to specify variable-length output fields. If this entry is blank for character, graphic, and UCS-2 data, then the external format is fixed length. For more information on variable-length fields, see "Variable-Length Character, Graphic, and UCS-2 Format" on page 113.

Note: The number of bytes occupied in the output record depends on the format specified. For example, a date written in *MDY format requires 8 bytes, but a date written in *ISO format requires 10 bytes.

For more information on external formats, see "Internal and External Formats" on page 103.

Externally Described Files

Externally described files include the following entries on input specifications.

Position 6 (Form Type)

An O must appear in position 6 to identify this line as an output specifications statement.

Record Identification and Control Entries

Output specifications for an externally described file are optional. Entries in positions 7 through 39 of the record identification line identify the record format and determine under what conditions the records are to be written.

Positions 7-16 (Record Name)

Entry	Explanation
A valid record format name	A record format name must be specified for an externally described file.

Positions 16-18 (External Logical Relationship)

Entry	Explanation
AND or OR	AND/OR indicates a relationship between lines of output indicators. AND/OR lines are valid for output records, but not for fields.

See "Positions 16-18 (Logical Relationship)" on page 323 for more information.

Position 17 (Type)

Entry	Explanation
E	Exception records.

Position 17 indicates the type of record to be written.

See "Position 17 (Type - Program Described File)" on page 323 for more information.

Positions 18-20 (Record Addition)

Entry	Explanation
ADD	Add a record to a file
DEL	Delete an existing record from the file

Positions 21-29 (Output Indicators)

Output indicators for externally described files are specified in the same way as those for program described files. For more information on output indicators, see "Positions 21-29 (File Record ID Indicators)" on page 324.

Positions 30-39 (EXCEPT Name)

An EXCEPT name can be specified in these positions for an exception record line. See "Positions 30-39 (EXCEPT Name)" on page 324 for more information.

Field Description and Control Entries

For externally described files, the only valid field descriptions are output indicators (positions 21 through 29), field name (positions 30 through 43), and blank after (position 45).

Positions 21-29 (Output Indicators)

Indicators specified on the field description lines determine whether a field is to be included in the output record. The same types of indicators can be used to control fields as are used to control records. See "Positions 21-29 (File Record ID Indicators)" on page 324 for more information.

Positions 30-43 (Field Name)

Entry	Explanation
Valid field name	A field name specified for an externally described file must be present in the external description unless the external name was renamed for the program.
***ALL**	Specifies the inclusion of all the fields in the record.

For externally described files, only the fields specified are placed in the output record. *ALL can be specified to include all the fields in the record. If *ALL is specified, no other field description lines can be specified for that record. In particular, you cannot specify a B (blank after) in position 45.

For an update record, only those fields specified in the output field specifications and meeting the conditions specified by the output indicators are placed in the output record to be rewritten. The values that were read are used to rewrite all other fields.

For the creation of a new record (ADD specified in positions 18-20), the fields specified are placed in the output record. Those fields not specified or not meeting the conditions specified by the output indicators are written as zeros or blanks, depending on the data format specified in the external description.

Position 45 (Blank After)

Entry	Explanation
Blank	The field is not reset.
B	The field specified in positions 30 through 43 is reset to blank, zero, or the default date/time/timestamp value after the output operation is complete.

Position 45 is used to reset a numeric field to zeros or a character, graphic, or UCS-2 field to blanks. Date, time, and timestamp fields are reset to their default values.

If the field is conditioned by indicators in positions 21 through 29, the blank after is also conditioned. This position must be blank for look-ahead, user date reserved words, *PLACE, named constants, and literals.

If blank after (position 45) is specified for a field to be written more than once, the B should be entered on the last line specifying output for that field, or else the field named is printed as the blank-after value for all lines after the one doing the blank after.

Chapter 22. Procedure Specifications

Procedure specifications are used to define prototyped procedures that are specified after the main source section, otherwise known as subprocedures.

The prototype for the subprocedure must be defined in the main source section of the module containing the subprocedure definition. A subprocedure includes the following:

1. A Begin-Procedure specification (B in position 24 of a procedure specification)
2. A Procedure-Interface definition, which specifies the return value and parameters, if any. The procedure-interface definition is optional if the subprocedure does not return a value and does not have any parameters that are passed to it. The procedure interface must match the corresponding prototype.
3. Other definition specifications of variables, constants and prototypes needed by the subprocedure. These definitions are local definitions.
4. Any calculation specifications needed to perform the task of the procedure. Any subroutines included within the subprocedure are local. They cannot be used outside of the subprocedure. If the subprocedure returns a value, then a RETURN operation must be coded within the subprocedure. You should ensure that a RETURN operation is performed before reaching the end of the procedure.
5. An End-Procedure specification (E in position 24 of a procedure specification)

Except for a procedure-interface definition, which may be placed anywhere within the definition specifications, a subprocedure must be coded in the order shown above.

For more information on the structure of the main source section and how the placement of definitions affects scope, see “Placement of Definitions and Scope” on page 256. See Chapter 6, “Subprocedures and Prototypes,” on page 63 for information on subprocedures and prototyping.

Procedure Specification Statement

The general layout for the procedure specification is as follows:
- The procedure specification type (P) is entered in position 6
- The non-commentary part of the specification extends from position 7 to position 80:
  - The fixed-format entries extend from positions 7 to 24
  - The keyword entries extend from positions 44 to 80
- The comments section of the specification extends from position 81 to position 100

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10

PName++++++++++..B..................Keywords++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 117. Procedure Specification Layout

Procedure Specification Keyword Continuation Line

If additional space is required for keywords, the keywords field can be continued on subsequent lines as follows:
- Position 6 of the continuation line must contain a P
- Positions 7 to 43 of the continuation line must be blank
- The specification continues on or past position 44

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
P.....................................Keywords++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 118. Procedure Specification Keyword Continuation Line Layout

Procedure Specification Continued Name Line

A name that is up to 15 characters long can be specified in the Name entry of the Procedure specification without requiring continuation. Any name (even one with 15 characters or fewer) can be continued on multiple lines by coding an ellipsis (...) at the end of the partial name.

A name definition consists of the following parts:
1. Zero or more continued name lines. Continued name lines are identified as having an ellipsis as the last non-blank characters in the entry. The name must begin within positions 7 - 21 and may end anywhere up to position 77 (with an ellipsis ending in position 80). There cannot be blanks between the start of the name and the ellipsis (...) characters. If any of these conditions is not true, the line is parsed as a main definition line.
2. One main definition line containing name, definition attributes, and keywords. If a continued name line is coded, the name entry of the main definition line may be left blank.
3. Zero or more keyword continuation lines.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10

DContinuedName+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 119. Procedure-Specification Continued Name Line

Position 6 (Form Type)

Enter a P in this position for a procedure specification

Positions 7-21 (Name)

Entry	Explanation
Name	The name of the subprocedure to be defined.

Use positions 7-21 to specify the name of the subprocedure being defined. If the name is longer than 15 characters, a name is specified in positions 7 - 80 of the continued name lines. The name can begin in any position in the space provided.

The name specified must be the same as the name of the prototype describing the procedure. If position 24 contains an E, then the name is optional.

Position 24 (Begin/End Procedure)

Entry	Explanation
B	The specification marks the beginning of the subprocedure being defined.
E	The specification marks the end of the subprocedure being defined.

A subprocedure coding consists minimally of a beginning procedure specification and an ending procedure specification. Any parameters and return value, as well as other definitions and calculations for the subprocedure are specified between the procedure specifications.

Positions 44-80 (Keywords)

Positions 44 to 80 are provided for procedure-specification keywords. Only a Begin-Procedure specification (B in position 24) can have a keyword entry.

Procedure Specification Keywords

Procedure specifications currently allow “EXPORT.”

EXPORT

The specification of the EXPORT keyword allows the procedure to be exported from a NOMAIN DLL. The name in positions 7-21 is exported in uppercase form.

If the EXPORT keyword is not specified, the procedure can only be called from within the module.

Part 4. Operations, Expressions, and Functions

This section describes the ways in which you can manipulate data or devices. The major topics include:

- Chapter 23, “Operations,” on page 341 provides an overview of operation codes grouped by function.
- Chapter 24, “Expressions,” on page 381 describes expressions and the rules governing them.
- “Built-In Functions (Alphabetically)” on page 405 describes built-in functions and their use on definition and calculation specifications.
- Chapter 26, “Operation Code Details,” on page 501 describes each operation code in detail.

Chapter 23. Operations

The VisualAge RPG programming language allows you to do many different types of operations on your data. To perform an operation, you use either an operation code or a built-in function.

This chapter summarizes the operation codes and built-in functions that are available. It also organizes the operation codes and built-in functions into categories.

For detailed information about a specific operation code or built-in function, see Chapter 26, "Operation Code Details," on page 501 or "Built-In Functions (Alphabetically)" on page 405.

Operation Codes

The following table shows the free-form syntax for each operation code.

- Extenders
  - **(D)** Date field
  - **(E)** Error handling
  - **(H)** Half adjust (round the numeric result)
  - **(M)** Default precision rules
  - **(N)** Do not lock record
  - **(N)** Set pointer to *NULL after successful DEALLOC
  - **(N)** Do not force data to non-volatile storage
  - **(P)** Pad the result with blanks or zeros
  - **(R)** "Result Decimal Position" precision rules
  - **(T)** Time field
  - **(Z)** Timestamp field

Table 32. Operation Codes in Free-Form Syntax

Code	Free-Form Syntax
BEGACT	BEGACT *action-subroutine-name*
BEGSR	BEGSR *subroutine-name*
CALLP	{CALLP{(EMR)}} *name*({*parm1*{:*parm2*...}})
CHAIN	CHAIN{(ENHMR)} *search-arg file-or-record-name* {*data-structure*}
CLEAR	CLEAR {*NOKEY} {*ALL} *name*
CLOSE	CLOSE{(E)} *file-name*
CLSWIN	CLSWIN{(E)} *window-name*
COMMIT	COMMIT{(E)}
DEALLOC	DEALLOC{(EN)} *pointer-name*
DELETE	DELETE{(EHMR)} {*search-arg*} *file-or-record-name*
DOU	DOU{(MR)} *indicator-expression*
DOW	DOW{(MR)} *indicator-expression*
DSPLY	DSPLY{(E)} *message* {*message-window-definition-name* | *DFT {*response*}}
ELSE	ELSE
ELSEIF	ELSEIF{(MR)} *indicator-expression*

Table 32. Operation Codes in Free-Form Syntax (continued)

Code	Free-Form Syntax
ENDACT	ENDACT {*return-point*}
ENDDO	ENDDO
ENDFOR	ENDFOR
ENDIF	ENDIF
ENDMON	ENDMON
ENDSL	ENDSL
ENDSR	ENDSR {*return-point*}
EVAL	{EVAL{(HMR)}} *result = expression*
EVALR	EVALR{(MR)} *result = expression*
EXCEPT	EXCEPT {*except-name*}
EXSR	EXSR *subroutine-name*
FEOD	FEOD{(EN)} *file-name*
FOR	FOR{(MR)} *index* {= *start*} {BY *increment*} {TO | DOWNTO *limit*}
IF	IF{(MR)} *indicator-expression*
IN	IN{(E)} {*LOCK} *data-area-name*
ITER	ITER
LEAVE	LEAVE
LEAVESR	LEAVESR
MONITOR	MONITOR
ON-ERROR	ON-ERROR {*exception-id1* {:*exception-id2*...}}
OPEN	OPEN{(E)} *file-name*
OTHER	OTHER
OUT	OUT{(E)} {*LOCK} *data-area-name*
POST	POST{(E)} *file-name*
READ	READ{(EN)} *name* {*data-structure*}
READC	READC{(E)} *subfile-name* {*subfile-index*}
READE	READE{(ENHMR)} *search-arg* | *KEY *file-or-record-name* {*data-structure*}
READP	READP{(EN)} *name* {*data-structure*}
READPE	READPE{(ENHMR)} *search-arg* | *KEY *file-or-record-name* {*data-structure*}
READS	READS{(E)} *subfile-name* {*subfile-index*}
RESET	RESET{(E)} {*NOKEY} {*ALL} *name*
RETURN	RETURN{(HMR)} *expression*
ROLBK	ROLBK{(E)}
SELECT	SELECT
SETGT	SETGT{(EHMR)} *search-arg file-or-record-name*
SETLL	SETLL{(EHMR)} *search-arg file-or-record-name*
SHOWWIN	SHOWWIN{(E)} *window-name*
SORTA	SORTA *array-name*
START	START{(E)} *name*
STOP	STOP{(E)} *component-name*

Table 32. Operation Codes in Free-Form Syntax (continued)

Code	Free-Form Syntax
TEST	TEST{(EDTZ)} {*dtz-format*} *field-name*
UNLOCK	UNLOCK{(E)} *name*
UPDATE	UPDATE{(E)} *file-or-record-name* {*data-structure* | %FIELDS(*name*{:*name*...})}
WHEN	WHEN{(MR)} *indicator-expression*
WRITE	WRITE{(E)} *name* {*data-structure*}

The next table is a summary of the specifications for each operation code in traditional syntax.

- An empty column indicates that the field must be blank
- All underlined fields are required
- An underscored space denotes that there is no resulting indicator in that position
- Symbols:
  - **+** Plus
  - **–** Minus
- Extenders:
  - **(D)** Date field
  - **(E)** Error handling
  - **(H)** Half adjust (round the numeric result)
  - **(M)** Default precision rules
  - **(N)** Do not lock record
  - **(P)** Pad the result with blanks or zeros
  - **(R)** ″Result Decimal Position″ precision rules
  - **(T)** Time field
  - **(Z)** Timestamp field
- Resulting indicator symbols:
  - **BL** Blank(s)
  - **BN** Blank(s) then numeric
  - **BOF** Beginning of the file
  - **EOF** End of the file
  - **EQ** Equal
  - **ER** Error
  - **FD** Found
  - **HI** Greater than
  - **IN** Indicator
  - **LO** Less than
  - **LR** Last record
  - **NR** No record was found
  - **NU** Numeric
  - **OF** Off
  - **ON** On
  - **Z** Zero
  - **ZB** Zero or Blank

Table 33. Operation Codes in Traditional Syntax

				Resulting Indicators		
Codes	**Factor 1**	**Factor 2**	**Result Field**	**71-72**	**73-74**	**75-76**
ADD (H)	Addend	Addend	Sum	+	–	Z
ADDDUR (E)	Date/Time	Duration:Duration Code	Date/Time		ER	

Table 33. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
ALLOC (E)		Length	Pointer		ER	
ANDxx	Comparand	Comparand				
BEGACT	Part name	Event name	Window name			
BEGSR	subroutine-name					
BITOFF		Bit numbers	Character field			
BITON		Bit numbers	Character field			
CABxx	Comparand	Comparand	Label	HI	LO	EQ
CALL (E)		Program name	Plist name		ER	
CALLB (E)		Procedure name or Procedure pointer	Plist name		ER	
CALLP (M/R)		name{ (parm1 {:parm2...}) }				
CASxx	Comparand	Comparand	Subroutine name	HI	LO	EQ
CAT (P)	Source string 1	Source string 2:number of blanks	Target string			
CHAIN (E N)	search-arg	name (file or record format)	data-structure	NR[2]	ER	
CHECK (E)	Comparator String	Base String:start	Left-most Position(s)		ER	FD[2]
CHECKR (E)	Comparator String	Base String:start	Right-most Position(s)		ER	FD[2]
CLEAR	*NOKEY	*ALL	name (variable, record format, or window)			
CLOSE (E)		file-name or *ALL			ER	
CLSWIN (E)		window-name			ER	
COMMIT (E)					ER	
COMP[1]	Comparand	Comparand		HI	LO	EQ
DEALLOC (E/N)			pointer-name		ER	
DEFINE	*LIKE	Referenced field	Defined field			
DEFINE	*DTAARA	External data area	Internal field			
DELETE (E)	search-arg, or subfile-index	name (file or record format)		NR[2]	ER	
DIV (H)	Dividend	Divisor	Quotient	+	–	Z
DO	Starting value	Limit value	Index value			
DOU (M/R)		indicator-expression				
DOUxx	Comparand	Comparand				
DOW (M/R)		indicator-expression				
DOWxx	Comparand	Comparand				
DSPLY (E)	message	message-window-definition name	response		ER	
ELSE						

Table 33. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
ELSEIF (M/R)		indicator-expression				
END		Increment value				
ENDACT		Return point				
ENDCS						
ENDDO		Increment value				
ENDOR						
ENDIF						
ENDMON						
ENDSL						
ENDSR	label	return-point				
EVAL (H M/R)		Result = Expression				
EVALR (M/R)		Result = Expression				
EXCEPT		except-name				
EXSR		subroutine-name				
EXTRCT (E)		Date/Time:Duration Code	Target Field		ER	
FEOD (EN)		file-name			ER	
FOR		Index-name = start-value BY increment TO|DOWNTO limit				
GETATR (E)	Part name	Attribute name	Field name		ER	
GOTO		Label				
IF (M/R)		indicator-expression				
IFxx	Comparand	Comparand				
IN (E)	*LOCK	data-area-name			ER	
ITER						
KFLD			Key field			
KLIST	KLIST name					
LEAVE						
LEAVESR						
LOOKUP[1] (array)	Search argument	Array name		HI	LO	EQ[5] on page 348
LOOKUP[1] (table)	Search argument	Table name	Table name	HI	LO	EQ[5] on page 348
MONITOR						
MOVE (P)	Data Attributes	Source field	Target field	+	–	ZB
MOVEA (P)		Source	Target	+	–	ZB
MOVEL (P)	Data Attributes	Source field	Target field	+	–	ZB
MULT (H)	Multiplicand	Multiplier	Product	+	–	Z
MVR			Remainder	+	–	Z
OCCUR (E)	Occurrence value	Data structure	Occurrence value		ER	
ON-ERROR		Status codes				

Table 33. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
OPEN (E)		file-name			ER	
ORxx	Comparand	Comparand				
OTHER						
OUT (E)	*LOCK	data-area-name			ER	
PARM	Target field	Source field	Parameter			
PLIST	PLIST name					
POST (E)[3]		file-name	INFDS name		ER	
READ (E N)		name (file, record format, or window)	Data structure		ER	EOF[4] on page 348
READC (E)		Subfile-name	Subfile-index		ER	EOF[4] on page 348
READE (E N)	search-argument	name (file or record format)	Data structure		ER	EOF[4] on page 348
READP (E N)		name (file or record format)	Data structure		ER	BOF[4] on page 348
READPE (E N)	Search argument	name (file or record format)	Data structure		ER	BOF[4] on page 348
READS (E)		Subfile-name	Subfile-index		ER	EOF[4] on page 348
REALLOC (E)		Length	Pointer		ER	
RESET (E)	*NOKEY	*ALL	name (variable, record format, or window)		ER	
RETURN (H M/R)		Expression				
ROLBK (E)					ER	
SCAN (E)	Comparator string:length	Base string:start	Left-most position(s)		ER	FD[2]
SELECT						
SETATR (E)	Part name	Attribute value	attribute		ER	
SETGT (E)	search-argument	name (file or record format)		NR[2]	ER	
SETLL (E)[5] on page 348	search-argument	name (file or record format)		NR[2]	ER	EQ
SETOFF[1]				OF	OF	OF
SETON[1]				ON	ON	ON
SHOWWIN (E)		window-name			ER	
SORTA		array-name				
SQRT (H)		Value	Root			

Table 33. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
START (E)		Component name or Field name	PLIST name		ER	
STOP (E)		Component name			ER	
SUB (H)	Minuend	Subtrahend	Difference	+	–	Z
SUBDUR (E) (duration)	Date/Time/ Timestamp	Date/Time/Timestamp	Duration: Duration Code		ER	
SUBDUR (E) (new date)	Date/Time/ Timestamp	Duration:Duration Code	Date/Time/ Timestamp		ER	
SUBST (E P)	Length to extract	Base string:start	Target string		ER	
TAG	Label					
TEST (E)[7] on page 348			Date/Time or Timestamp Field		ER	
TEST (D E)[7] on page 348	Date Format		Character or Numeric field		ER	
TEST (E T)[7] on page 348	Time Format		Character or Numeric field		ER	
TEST (E Z)[7] on page 348	Timestamp Format		Character or Numeric field		ER	
TESTB[1]		Bit numbers	Character field	OF	ON	EQ
TESTN[1]			Character field	NU	BN	BL
TESTZ[1]			Character field	+	–	
TIME	Alias name		Target field			
UNLOCK (E)		name (file or data area)			ER	
UPDATE (E)		name (file, record format, or window)	Data structure		ER	
WHEN (M/R)		indicator-expression				
WHENxx	Comparand	Comparand				
WRITE (E)		name (file, record format, subfile, or window)	Data structure		ER	EOF[4] on page 348
XFOOT (H)		Array name	Sum	+	–	Z
XLATE (E P)	From:To	String:start	Target String		ER	
Z-ADD (H)		Addend	Sum	+	–	Z
Z-SUB (H)		Subtrahend	Difference	+	–	Z

Table 33. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76

Notes:

1. At least one resulting indicator is required.
2. The %FOUND built-in function can be used as an alternative to specifying an NR or FD resulting indicator.
3. You must specify factor 2 or the result field. You may specify both.
4. The %EOF built-in function can be used as an alternative to specifying an EOF or BOF resulting indicator.
5. The %EQUAL built-in function can be used to test the SETLL and LOOKUP operations.
6. For all operation codes with extender 'E', either the extender 'E' or an ER error indicator can be specified, but not both.
7. You must specify the extender 'E' or an error indicator for the TEST operation.

Arithmetic Operations

The arithmetic operations are shown in the following table.

Table 34. Arithmetic Operations

Operation	Traditional Syntax	Free-Form Syntax
Absolute Value	"%ABS (Absolute Value of Expression)" on page 405	
Add	"ADD (Add)" on page 501	+ operator
Divide	"DIV (Divide)" on page 553	/ operator or "%DIV (Return Integer Portion of Quotient)" on page 431
Division Remainder	"MVR (Move Remainder)" on page 636	"%REM (Return Integer Remainder)" on page 465
Multiply	"MULT (Multiply)" on page 635	* operator
Square Root	"SQRT (Square Root)" on page 688	"%SQRT (Square Root of Expression)" on page 474
Subtract	"SUB (Subtract)" on page 692	- operator
Zero and Add	"Z-ADD (Zero and Add)" on page 722	(not allowed)
Zero and Subtract	"Z-SUB (Zero and Subtract)" on page 723	(not allowed)

For examples of arithmetic operations, see Figure 120 on page 351.

The following rules apply when specifying arithmetic operations:

- Arithmetic operations can be done only on numerics:
  - numeric subfields
  - numeric arrays
  - numeric array elements
  - numeric table elements
  - numeric named constants
  - numeric figurative constants
  - numeric literals
- In general, arithmetic operations are performed using the packed-decimal format. This means that the fields are first converted to packed-decimal format prior to performing the arithmetic operation, and then converted back to their specified format (if necessary) prior to placing the result in the result field. However, note the following exceptions:
  - If all operands are unsigned, the operation will use unsigned arithmetic.

  - If all are integer, or integer and unsigned, then the operation will use integer arithmetic.
  - If any operands are float, then the remaining operands are converted to float.

  However, the DIV operation uses either the packed-decimal or float format for its operations. For more information on integer and unsigned arithmetic, see "Integer and Unsigned Arithmetic"
- Decimal alignment is done for all arithmetic operations. Even though truncation can occur, the position of the decimal point in the result field is not affected.
- The length of any field specified in an arithmetic operation cannot exceed 31 digits. If the result exceeds 31 digits, digits are dropped from either or both ends, depending on the location of the decimal point.
- The TRUNCNBR option determines whether truncation on the left occurs with numeric overflow or a runtime error is generated. This option can be specified on the Build window. For more information, see *Getting Started with WebSphere Development Studio Client for iSeries* .
- An arithmetic operation does not change factor 1 and factor 2 unless they are the same as the result field.
- The result of an arithmetic operation replaces the data that was in the result field.
- Half-adjusting is done by adding 5 (-5 if the field is negative) one position to the right of the last specified decimal position in the result field. The half adjust entry is allowed only with arithmetic operations, but not with an MVR operation or with a DIV operation followed by the MVR operation. Half adjust only affects the result if the number of decimal positions in the calculated result is greater than the number of decimal positions in the result field. Half adjusting occurs after the operation but before the result is placed in the result field. Resulting indicators are set according to the value of the result field after half-adjusting has been done.
- If you use conditioning indicators with DIV and MVR, it is your responsibility to ensure that the DIV operation occurs immediately before the MVR operation. If conditioning indicators on DIV cause the MVR operation to be executed when the immediately preceeding DIV was not executed, then undesirable results may occur.

For arithmetic operations in which all three fields are used:
- Factor 1, factor 2, and the result field can be three different fields
- Factor 1, factor 2, and the result field can all be the same field
- Factor 1 and factor 2 can be the same field but different from the result field
- Either factor 1 or factor 2 can be the same as the result field.

For information on using arrays with arithmetic operations, see "Specifying an Array in Calculations" on page 183.

Performance Considerations

The fastest performance time for arithmetic operations occurs when all operands are in integer or unsigned format. The next fastest performance time occurs when all operands are in packed format, since this eliminates conversions to a common format.

Integer and Unsigned Arithmetic

For all arithmetic operations (not including those in expressions) if factor 1, factor 2, and the result field are defined with unsigned format, then the operation is performed using unsigned format. Similarly, if factor 1, factor 2, and the result field are defined as either integer or unsigned format, then the operation is

performed using integer format. If any field does not have either integer or unsigned format, then the operation is performed using the default format, packed-decimal.

The following points apply to integer and unsigned arithmetic operations only:

- If any of the fields are defined as 4-byte fields, then all fields are first converted to 4 bytes before the operation is performed.
- Integer and unsigned values may be used together in one operation. However, if either factor 1, factor 2, or the result field is integer, then all unsigned values are converted to integer. If necessary, a 1-byte, 2-byte, or 4-byte unsigned value is converted to a larger-sized integer value to lessen the chance of numeric overflow.
- If a literal has 20 digits or less with zero decimal positions, and falls within the range allowed for integer and unsigned fields, then it is loaded in integer or unsigned format, depending on whether it is a negative or positive value respectively.

Note: Integer or unsigned arithmetic may give better performance. However, the chances of numeric overflow may be greater when using integer or unsigned numeric format, than when using packed or zoned decimal format.

Arithmetic Operations Examples

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
C*
C*   In the following example, the initial field values are:
C*
D A               s              3p 0  inz(1)
D B               s              3p 1  inz(10.0)
D C               s              2p 0  inz(32)
D D               s              2p 0 inz(-10)
D E               s              3p 0 inz(6)
D F               s              3p 0 inz(10)
D G               s              3p 2 inz(2.77)
D H               s              3p 0 inz(70)
D J               s              3p 1 inz(0.6)
D K               s              2p 0 inz(25)
D L               s              2p 1 dim(3)
D V               s              5p 2
D W               s              5p 1
D X               s              8p 4
D Y               s              6p 2
D Z               s              5p 3

 /FREE
    L(1) = 1.0;
    L(2) = 1.7;
    L(3) = -1.1;

    A = A + 1;               // A = 002
    V = B + C;               // V = 042.00
    V = B + D;               // V = 0
    V = C;                   // V = 032.00
    E = E - 1;               // E = 005
    W = C - B;               // W = 0022.0
    W = C - D;               // W = 0042.0
    W = - C;                 // W = -0032.0
    F = F * E;               // F = 060
    X = B * G;               // X = 0027.7000
    X = B * D;               // X = -0100.0000
    H = H / B;               // H = 007
    Y = C / J;               // Y = 0053.33
    eval(r) Z = %sqrt(K);    // Z = 05.000
    Z = %xfoot(L);           // Z = 01.600

    dump(a);
    *inlr = *on;
 /END-FREE
```

Figure 120. Summary of Arithmetic Operations

Array Operations

The array operations are shown in the following table.

Table 35. Array Operations

Operation	Traditional Syntax	Free-Form Syntax
Look Up Elements	"LOOKUP (Look Up a Table or Array Element)" on page 599	"%LOOKUPxx (Look Up an Array Element)" on page 455 or "%TLOOKUPxx (Look Up a Table Element)" on page 489
Number of Elements	"%ELEM (Get Number of Elements)" on page 437	
Move an Array	"MOVEA (Move Array)" on page 619	(not allowed)

Table 35. Array Operations (continued)

Operation	Traditional Syntax	Free-Form Syntax
Sort an Array	"SORTA (Sort an Array)" on page 686	
Subset an Array	"%SUBARR (Set/Get Portion of an Array)" on page 480	
Sum the Elements of an Array	"XFOOT (Summing the Elements of an Array)" on page 719	"%XFOOT (Sum Array Expression Elements)" on page 497

While many operations work with arrays, these operations perform specific array functions. See each operation for an explanation of its function.

Bit Operations

The bit operations are:
- "%BITAND (Bitwise AND Operation)" on page 409
- "%BITNOT (Invert Bits)" on page 410
- "%BITOR (Bitwise OR Operation)" on page 411
- "%BITXOR (Bitwise Exclusive-OR Operation)" on page 412
- "BITOFF (Set Bits Off)" on page 512
- "BITON (Set Bits On)" on page 513
- "TESTB (Test Bit)" on page 703.

Table 36. Bit Operations

Operation	Traditional Syntax	Free-Form Syntax
Set bits on	BITON	%BITOR
Set bits off	BITOFF	%BITAND with %BITNOT
Test bits	TESTB	%BITAND (see example of Figure 138 on page 413)

The bits in a byte are numbered from left to right. The left most bit is bit number 0. In these operations, factor 2 specifies the bit pattern (bit numbers) and the result field specifies a one-byte character field on which the operation is performed. To specify the bit numbers in factor 2, a 1-byte hexadecimal literal or a 1-byte character field is allowed. The bit numbers are indicated by the bits that are turned on in the literal or the field. Alternatively, a character literal which contains the bit numbers can also be specified in factor 2.

With the BITAND operation the result bit is ON when all of the corresponding bits in the arguments are ON, and OFF otherwise.

With the BITNOT operation the result bit is ON when the corresponding bit in the argument is OFF, and OFF otherwise.

With the BITOR operation the result bit is ON when any of the corresponding bits in the arguments are ON, and OFF otherwise.

With the BITXOR operation the result bit is ON when just one of the corresponding bits in the arguments are ON, and OFF otherwise.

Branching Operations

The branching operations are shown in the following table.

Table 37. Branching Operations

Operation	Traditional Syntax	Free-Form Syntax
Compare and Branch	"CABxx (Compare and Branch)" on page 515	(not allowed)
Go To	"GOTO (Go To)" on page 585	(not allowed)
Iterate	"ITER (Iterate)" on page 591	
Leave	"LEAVE (Leave a Do/For Group)" on page 596	
Leave a subroutine	"LEAVESR (Leave a Subroutine)" on page 598	
Tag	"TAG (Tag)" on page 699	(not allowed)

See each operation for an explanation of its function.

Call Operations

The call operations are shown in the following table.

Table 38. Call Operations

Operation	Traditional Syntax	Free-Form Syntax
Call Program or Procedure	• "CALL (Call an AS/400 Program)" on page 517 • "CALLB (Call a Function)" on page 521 • "CALLP (Call a Prototyped Procedure or Program)" on page 522	"CALLP (Call a Prototyped Procedure or Program)" on page 522
Identify Parameters	• "PARM (Identify Parameters)" on page 647 • "PLIST (Identify a Parameter List)" on page 650	PI or PR definition specification
Return	"RETURN (Return to Caller)" on page 671	
Start a Component	"START (Start Component or Call Local Program)" on page 689	

See each operation for an explanation of its function.

CALLP is one of type of prototyped call. The second type is a call from within an expression. A **prototyped call** is a call for which there is a prototype defined for the call interface.

Call operations allow a VisualAge RPG procedure to transfer control to other programs or procedures. However, prototyped calls differ from the CALL and CALLB operations in that they allow free-form syntax.

The RETURN operation transfers control back to the calling program or procedure and returns a value, if any. The PLIST and PARM operations can be used with the CALL and CALLB operations to indicate which parameters should be passed on the call. With a prototyped call, you pass the parameters on the call.

The recommended way to call a program or procedure (written in any language) is to code a protyped call.

Prototyped Calls

With a prototyped call, you can call (with the same syntax):
- Programs that are on the system at run time
- Exported procedures in other modules that are bound in the same program

- Subprocedures in the same module

A prototype must be included in the definition specifications of the program or procedure making the call. It is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters.

When a program or procedure is prototyped, you do not need to know the names of the data items used in the program or procedure; only the number and type of parameters.

Prototypes improve the communication between programs and procedures. Some advantages of using prototypes calls are:
- The syntax is simplified because no PARM or PLIST operations are required.
- For some parameters, you can pass literals and expressions.
- The compiler helps you pass enough parameters, of the correct type, format and length, by giving an error at compile time if the call is not correct.
- The compiler helps you pass enough parameters with the correct format and length for some types of parameters, by doing a conversion at run-time.

Figure 121 shows an example using the prototype ProcName, passing three parameters. The prototype ProcName could refer to either a program or procedure. It is not important to know this when making the call; this is only importantwhen defining the prototype.

```
/FREE
// The following calls ProcName with the 3
// parameters CharField, 7, and Field2:
     ProcName (CharField: 7: Field2);

// If you need to specify operation extenders, you must also
// specify the CALLP operation code:
     CALLP(e) ProcName (CharField: 7: Field2);
/END-FREE
```

Figure 121. Sample of CALLP operation

When calling a procedure in an expression, you should use the procedure name in a manner consistent with the data type of the specified return value. For example, if a procedure is defined to return a numeric, then the call to the procedure within an expression must be where a numeric would be expected.

For more information on calling programs and procedures, and passing parameters, see *Programming with VisualAge RPG*. For more information on defining prototypes and parameters, see "Prototypes and Parameters" on page 71.

Parsing Program Names on a Call

Program names are specified in factor 2 of a CALL operation. If you specify the library name, it must be immediately followed by a slash and then the program name (for example, 'LIB/PROG'). If a library is not specified, the library list is used to find the program. *CURLIB is not supported.

Note the following rules:
- The total length of a literal, including the slash, cannot exceed 12 characters.
- The total length of the non-blank data in a field or named constant, including the slash, cannont exceed 21 characters.

- If either the program or the library name exceeds 10 characters, it is truncated to 10 characters.

The program name is used exactly as specified in the literal, field, named constant, or array element to determine the program to be called. Specifically:
- Any leading or trailing blanks are ignored.
- If the first character in the entry is a slash, the library list is used to find the program.
- If the last character in the entry is a slash, a compile-time message will be issued.
- Lowercase characters are not shifted to uppercase.
- A name enclosed in quotation marks, for example, '"ABC"', always includes the quotation marks as part of the name of the program to be called.

Program references are grouped to avoid the overhead of resolving to the target program. All references to a specific program using a named constant or literal are grouped so that the program is resolved to only once, and all subsequent references to that program (by way of named constant or literal only) do not cause a resolve to recur.

The program references are grouped if both the program and the library name are identical. All program references by variable name are grouped by the variable name. When a program reference is made with a variable, its current value is compared to the value used on the previous program reference operation that used that variable. If the value did not change, no resolve is done. If it did change, a resolve is done to the new program specified. Note that this rule applies only to references using a variable name. References using a named constant or literal are never resolved again, and they do not affect whether or not a program reference by variable is resolved again. illustrates the grouping of program references.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D Pgm_Ex_A        C                   'LIB1/PGM1'
D Pgm_Ex_B        C                   'PGM1'
D PGM_Ex_C        C                   'LIB/PGM2'
D*
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C                   CALL      Pgm_Ex_A
C*
C* The following two calls will be grouped together because both
C* have the same program name (PGM1) and the same library name
C* (none).  Note that these will not be grouped with the call using
C* Pgm_Ex_A above because Pgm_Ex_A has a different library
C* name specified (LIB1).
C*
C                   CALL      'PGM1'
C                   CALL      Pgm_Ex_B
C*
C* The following two program references will be grouped together
C* because both have the same program name (PGM2) and the same
C* library name (LIB).
C*
C                   CALL      'LIB/PGM2'
C                   CALL      Pgm_Ex_C
C*
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The first call in the program using CALLV below will result in
C* a resolve being done for the variable CALLV to the program PGM1.
C* This is independent of any calls by a literal or named constant
C* to PGM1 that may have already been done in the program.  The
C* second call using CALLV will not result in a resolve to PGM1
C* because the value of CALLV has not changed.
C*
C                   MOVE     'PGM1'         CALLV         21
C                   CALL     CALLV
C                   CALL     CALLV
```

Figure 122. Eample of Grouping or Program References

Compare Operations

The compare operations are shown in the following table.

Table 39. Compare Operations

Operation	Traditional Syntax	Free-Form Syntax
And	"ANDxx (And)" on page 506	AND operator
Compare	"COMP (Compare)" on page 545	=, <, >, <=, >=, or <> operator
Compare and Branch	"CABxx (Compare and Branch)" on page 515	(not allowed)
Conditional Subroutine	"CASxx (Conditionally Invoke Subroutine)" on page 524	"IF (If)" on page 586 and "EXSR (Invoke User Subroutine)" on page 577
Do Until	"DOU (Do Until)" on page 556 or "DOUxx (Do Until)" on page 557	"DOU (Do Until)" on page 556
Do While	"DOW (Do While)" on page 559 or "DOWxx (Do While)" on page 560	"DOW (Do While)" on page 559
If	"IF (If)" on page 586 or "IFxx (If)" on page 587	"IF (If)" on page 586
Or	"ORxx (Or)" on page 644	OR operator
When	"WHEN (When True Then Select)" on page 713 or "WHENxx (When True Then Select)" on page 714	"WHEN (When True Then Select)" on page 713

For the ANDxx, CABxx, CASxx, DOUxx, DOWxx, IFxx, ORxx, and WHENxx operations, xx can be:

xx	Meaning
GT	Factor 1 is greater than factor 2.
LT	Factor 1 is less than factor 2.
EQ	Factor 1 is equal to factor 2.
NE	Factor 1 is not equal to factor 2.
GE	Factor 1 is greater than or equal to factor 2.
LE	Factor 1 is less than or equal to factor 2.
Blanks	Unconditional processing (CASxx or CABxx).

The compare operations test fields for the conditions specified in the operations. These operations do not change the values of the fields. For COMP, CABXX, and CASXX, the resulting indicators assigned in positions 71 and 76 are set according to the results of the operation. All data types may be compared to fields of the same data type.

Remember the following when using the compare operations:

- If numeric fields are compared, fields of unequal length are aligned at the implied decimal point. The fields are filled with zeros to the left and/or right of the decimal point making the field lengths and number of decimal positions equal for comparison.
- All numeric comparisons are algebraic. A plus (+) value is always greater than a minus (-) value.
- All graphic and UCS-2 comparisons are done using the hexadecimal representation of the graphic characters.

- If character, graphic, or UCS-2 fields are compared, fields of unequal length are aligned to their leftmost character. The shorter field is filled with blanks to equal the length of the longer field so that the field lengths are equal for comparison.
- Date fields are converted to a common format when being compared.
- Time fields are converted to a common format when being compared.
- When basing pointer fields are compared for anything except equality or inequality, the results will be unpredictable unless the pointers point to addresses within contiguous storage (for example, all point to positions within the same data structure, array, or standalone field).
- When procedure pointer fields are compared for anything except equality or inequality, the results are unpredictable.
- An array name cannot be specified in a compare operation, but an array element may be specified.
- The ANDxx and ORxx operations can be used following DOUxx, DOWxx, IFxx, and WHENxx.
- When comparing a character, graphic, or UCS-2 literal with zero length to a field (fixed or varying) containing blanks, the fields will compare equal. If you want to test that a value is of length 0, use the %LEN built-in function.

Conversion Operations

The following built-in functions perform conversion operations:
- "%CHAR (Convert to Character Data)" on page 416
- "%DEC (Convert to Packed Decimal Format)" on page 424
- "%DECH (Convert to Packed Decimal Format with Half Adjust)" on page 426
- "%EDITC (Edit Value Using an Editcode)" on page 432
- "%EDITFLT (Convert to Float External Representation)" on page 435
- "%EDITW (Edit Value Using an Editword)" on page 436
- "%FLOAT (Convert to Floating Format)" on page 443
- "%GRAPH (Convert to Graphic Value)" on page 447
- "%INT (Convert to Integer Format)" on page 449
- "%INTH (Convert to Integer Format with Half Adjust)" on page 449
- "%UCS2 (Convert to UCS-2 Value)" on page 494
- "%UNS (Convert to Unsigned Format)" on page 495
- "%UNSH (Convert to Unsigned Format with Half Adjust)" on page 495

These built-in functions are available in both the traditional syntax and free-form syntax.

The traditional MOVE and MOVEL operation codes perform conversions when factor 2 and the result field have different types. See:
- "MOVE (Move)" on page 604
- "MOVEL (Move Left)" on page 626

Data-Area Operations

The data-area operations are:
- "IN (Retrieve a Data Area)" on page 589
- "OUT (Write a Data Area)" on page 646
- "UNLOCK (Unlock a Data Area or Release a Record)" on page 709.

These operations are available in both the traditional syntax and free-form syntax.

If your application accesses an OS/400 data area, the name of this data area can either be the name of the OS/400 data area or an override name that you defined

using the Define server information menu item. For more information on using the GUI Designer to define server information, see *Programming with VisualAge RPG*.

The following lock states are used:

IN operation with *LOCK	An exclusive allow read lock state is placed on the data area
OUT operation with *LOCK	The data area remains locked after the write operation
OUT operation with blank	The data area is unlocked after it is updated
UNLOCK	The data area is unlocked, the record locks are released, and the data areas and/or the records are not updated.

When data is moved into and out of a data area, the system locks the data area. If several users are contending for the same data area, a user may get an error message indicating that the data area is not available.

The following rules apply to data area operations:

- A data-area operation cannot be done on a data area that is not defined to the operating system.
- Before a data area operation can be done, the data area must be specified on the definition specification or on the DEFINE operation code. For more information, see "DEFINE (Field Definition)" on page 548.
- A locked data area cannot be updated or locked by another program.
- A data-area name cannot be the name of a multiple-occurrence data structure, an input record field, an array, an array element, or a table.
- A data area cannot be the subfield of a multiple occurrence data structure, a data-area data structure, a program-status data structure, a file-information data structure (INFDS), or a data structure that appears on an *DTAARA DEFINE statement.
- If the name of the data area is determined at runtime, due to the DTAARA(*VAR) keyword being used, the variable containing the name must be set before an IN operation. If a data area is locked because of a prior *LOCK IN operation, any other operations (IN, OUT, UNLOCK) for the data area will use the previously locked data area, and the variable containing the name will not be consulted.
- If the library name is not specified by the DTAARA keyword, the library list will be used to locate the data area.

Date Operations

The date operations are shown in the following table.

Table 40. Date Operations

Operation	Traditional Syntax	Free-Form Syntax
Add Duration	"ADDDUR (Add Duration)" on page 502	+ operator
Extract	"EXTRCT (Extract Date/Time/Timestamp)" on page 579	"%SUBDT (Extract a Portion of a Date, Time, or Timestamp)" on page 483
Subtract Duration	"SUBDUR (Subtract Duration)" on page 693	- operator or "%DIFF (Difference Between Two Date, Time, or Timestamp Values)" on page 428

Table 40. Date Operations (continued)

Operation	Traditional Syntax	Free-Form Syntax
Convert date/time/timestamp to character	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	"%CHAR (Convert to Character Data)" on page 416
Convert date/time/timestamp to numeric	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	"%DEC (Convert to Packed Decimal Format)" on page 424
Convert character/numeric to date	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	"%DATE (Convert to Date)" on page 422
Convert character/numeric to time	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	"%TIME (Convert to Time)" on page 487
Convert character/numeric/date to timestamp	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	"%TIMESTAMP (Convert to Timestamp)" on page 488
Move date/time to timestamp	"MOVE (Move)" on page 604 or "MOVEL (Move Left)" on page 626	date + time
Test	"TEST (Test Date/Time/Timestamp)" on page 700	
Number of Years	"%YEARS (Number of Years)" on page 499	
Number of Months	"%MONTHS (Number of Months)" on page 458	
Number of Days	"%DAYS (Number of Days)" on page 423	
Number of Hours	"%HOURS (Number of Hours)" on page 448	
Number of Minutes	"%MINUTES (Number of Minutes)" on page 457	
Number of Seconds	"%SECONDS (Number of Seconds)" on page 470	
Number of Microseconds	"%MSECONDS (Number of Microseconds)" on page 459	

Date operations allow you to work with dates, times, and timestamp fields and character or numeric fields that represent dates, times, and timestamps. You can:

- Add or subtract a duration in years, months, days, hours, minutes, seconds, or microseconds
- Determine the duration between two dates, times, or timestamps
- Extract a portion of a date, time, or timestamp (for example, the day)
- Test that a value is valid as a date, time, or timestamp.

To add or subtract a duration, you can use the + or - operator in free-form syntax or the ADDDUR or SUBDUR operation code in traditional syntax. The following table shows the built-in functions that you use in free-form syntax and the duration codes that you use in traditional syntax.

Table 41. Built-In Functions and Duration Codes

Unit	Built-In Function	Duration Code
Year	%YEARS	*YEARS or *Y
Month	%MONTHS	*MONTHS or *M
Day	%DAYS	*DAYS or *D
Hour	%HOURS	*HOURS or *H
Minute	%MINUTES	*MINUTES or *MN
Second	%SECONDS	*SECONDS or *S
Microsecond	%MSECONDS	*MSECONDS or *MS

For example, you can add 23 days to an existing date in either of the following ways:

```
C                   ADDDUR    23:*D         DUEDATE

 /FREE
     newdate = duedate + %DAYS(23)
 /END-FREE
```

To calculate the duration between two dates, times, or timestamps, you can use the %DIFF built-in function in free-form syntax or the SUBDUR operation code in traditional syntax. In either case, you must specify one of the duration codes shown in Table 41 on page 360.

The duration is given in complete units, with any remainder discarded. A duration of 59 minutes, expressed in hours, is 0. A duration of 61 minutes, expressed in hours, is 1.

The following table shows additional examples, using the SUBDUR operation code. The %DIFF built-in function would give the same results.

Table 42. Resulting Durations Using SUBDUR

Duration Unit	Factor 1	Factor 2	Result
Months	1999-03-28	1999-02-28	1 month
	1999-03-14	1998-03-15	11 months
	1999-03-15	1998-03-15	12 months
Years	1999-03-14	1998-03-15	0 years
	1999-03-15	1998-03-15	1 year
	1999-03-14-12.34.45.123456	1998-03-14-12.34.45.123457	0 years
Hours	1990-03-14-23.00.00.000000	1990-03-14-22.00.00.000001	0 hours

Unexpected Results

A month can contain 28, 29, 30, or 31 days. A year can contain 365 or 366 days. Because of this inconsistency, the following operations can give unexpected results:

- Adding or subtracting a number of months (or calculating a duration in months) with a date that is on the 29th, 30th, or 31st of a month
- Adding or subtracting a number of years (or calculating a duration in years) with a February 29 date.

The following rules are used:

- When months or years are added or subtracted, the day portion remains unchanged if possible. For example, 2000-03-15 + %MONTHS(1) is 2000-04-15.
- If the addition or subtraction would produce a nonexistent date (for example, April 31), the last day of the month is used instead.
- Any month or year operation that changes the day portion is **not** reversible. For example, `2000-03-31 + %MONTHS(1) is 2000-04-30` changes the day from 31 to 30. You cannot get back the original `2000-03-31` by subtracting one month.

  The operation `2000-03-31 + %MONTHS(1) - %MONTHS(1)` becomes `2000-03-30`.
- The duration between two dates is one month if the later date minus one month gives the first date. For example, the duration in months (rounded down) between 2000-03-31 and 2000-04-30 is 0 because 2000-04-30 - %MONTHS(1) is 2000-03-30 (not 2000-03-31).

Declarative Operations

The declarative operations are shown in the following table.

Table 43. Declarative Operations

Operation	Traditional Syntax	Free-Form Syntax
Define Field	"DEFINE (Field Definition)" on page 548	LIKE or DTAARA keyword on definition specification
Define Key	• "KFLD (Define Parts of a Key)" on page 593 • "KLIST (Define a Composite Key)" on page 594	(not allowed)
Identify Parameters	• "PARM (Identify Parameters)" on page 647 • "PLIST (Identify a Parameter List)" on page 650	PR definition specification
Tag	"TAG (Tag)" on page 699	(not allowed)

The declarative operations are used to declare the properties of fields or to mark parts of a program. The control level entry (positions 7 and 8) can be blank or can contain an entry to group the statements within the appropriate section of the program.

Error-Handling Operations

The exception-handling operation codes are:
- "MONITOR (Begin a Monitor Group)" on page 602
- "ON-ERROR (On Error)" on page 641
- ENDMON, as described in "ENDyy (End a Structured Group)" on page 566

These operation codes are available in both the traditional syntax and free-form syntax.

MONITOR, ON-ERROR and ENDMON are used to code a monitor group. The monitor group consists of a monitor block, followed by one or more on-error blocks, followed by ENDMON.

The monitor block contains the code that you think might generate an error. The on-error blocks contain the code to handle errors that occur in the monitor block.

A monitor block consists of a MONITOR operation followed by the operations that will be monitored. An on-error block consists of an ON-ERROR operation, with a list of status codes, followed by the operations that will be performed if an error in the monitor block generates any of the listed status codes.

When an error occurs in the monitor block and the operation has an (E) extender or an error indicator, the error will be handled by the (E) extender or the error indicator. If no indicator or extender can handle the error, control passes to the on-error block containing the status code for the error. When the on-error block is finished, control passes to the ENDMON. If there is no on-error block to handle the error, control passes to the next level of exception handling (the *PSSR or INFSR subroutines, or the default error handler).

```
/free
   MONITOR;                                _
       OPEN   FILE;                         |
       DOW    getNextRecord ();             |
         X = X + 1;                         +-- This is the monitor block
         nameList(X) = name;                |
       ENDDO;                               |
       CLOSE  FILE;                        _|
   ON-ERROR   1216;                        _
       DSPMSG                               |
          ('Error opening file FILE'        |
         : %status);                        +-- First on-error block
       RETURN;                             _|
   ON-ERROR   121;                         _
       DSPMSG                               |
          ('Array NAME is too small'        +-- Second on-error block
         : %status);                        |
       RETURN;                             _|
   ON-ERROR   *ALL;                        _
       DSPMSG                               |
          ('Unexpected error'               +-- Final catch-all on-error block
         : %status);                        |
       RETURN;                             _|
   ENDMON;                                 --- End of MONITOR group
/end-free
```

Figure 123. Example of MONITOR and ON-ERROR blocks

File Operations

The file operation codes are:

- "CHAIN (Random Retrieval from a File)" on page 529
- "CLOSE (Close Files)" on page 542
- "COMMIT (Commit)" on page 544
- "DELETE (Delete Record)" on page 551
- "EXCEPT (Calculation Time Output)" on page 575
- "FEOD (Force End of Data)" on page 580
- "OPEN (Open File for Processing)" on page 642
- "POST (Post)" on page 652
- "READ (Read a Record)" on page 653
- "READC (Read Next Changed Record)" on page 656
- "READE (Read Equal Key)" on page 658
- "READP (Read Prior Record)" on page 661
- "READPE (Read Prior Equal)" on page 663
- "READS (Read Selected)" on page 666
- "ROLBK (Roll Back)" on page 672
- "SETGT (Set Greater Than)" on page 679
- "SETLL (Set Lower Limit)" on page 681
- "UNLOCK (Unlock a Data Area or Release a Record)" on page 709
- "UPDATE (Modify Existing Record)" on page 711
- "WRITE (Create New Records)" on page 717

The file built-in functions are:

- "%EOF (Return End or Beginning of File Condition)" on page 438
- "%EQUAL (Return Exact Match Condition)" on page 440
- "%FOUND (Return Found Condition)" on page 444
- "%OPEN (Return File Open Condition)" on page 462

- "%STATUS (Return File or Program Status)" on page 475

These operations are available in both the traditional syntax and free-form syntax.

You can use the file operations to work with either local files, remote files, or with parts on a window:

Operation	OS/400 file	Local file	Subfile	Static text	Entry field	Special files
CHAIN	Y	Y	Y			
CLOSE	Y	Y				Y
COMMIT	Y					
DELETE	Y	Y	Y			Y
EXCEPT	Y	Y				
FEOD	Y					Y
OPEN	Y	Y				Y
POST	Y	Y				
READ	Y	Y		Y	Y	Y
READC			Y			
READE	Y					
READP	Y	Y				
READPE	Y					
READS			Y			
SETGT	Y					
SETLL	Y					
UNLOCK	Y					
UPDATE	Y	Y	Y			Y
WRITE	Y	Y	Y	Y	Y	Y

When an externally described file is used with certain file operations, a record format name, rather than a file name, can be specified in factor 2. Thus, the processing operation code retrieves and/or positions the file at a record format of the specified type according to the rules of the calculation operation code used.

The CHAIN, READ, READC, READE, READP, and READPE operations *may* have a result data structure. For these operations, data is transferred directly between the file and the data structure, without processing the input specifications for the file. Thus, no record identifying or field indicators are set on as a result of an input operation to a data structure. If all input operations to the file have a result data structure, input specifications are not required.

The WRITE and UPDATE operations that specify a program described file name in factor 2 *must* have a data structure name specified in the result field. WRITE and UPDATE operations to an externally described file or record *may* have a result data structure. For these operations, data is transferred directly between data structure and the file, without processing the output specifications for the file. If all output operations to the file have a result data structure, output specifications are not required.

A data structure name is allowed as the result of an I/O operation to an externally described file name or record name as follows:

1. When a record name is specified on an I/O operation, the origin of the data structure must match the record. That is, the data structure must be defined using LIKEREC(rec) or EXTNAME(file:rec) where rec is the format name specified on the operation. For input operations, the result data structure (or base structure for the LIKEDS data structure) must be defined using *INPUT. For output operations, the result data structure must be defined using *OUTPUT. For UPDATE to a DISK file, the result data structure may be defined using either *INPUT or *OUTPUT.
2. A result data structure may be specified for an I/O operation to an externally described file name, in addition to a record name, for opcodes CHAIN, READ, READE, READP, and READPE. When the name of an externally described file is specified, the data structure must contain one subfield data structure for each record with input-capable fields, where the allowed subfield data structures are defined as in rule 1. Each subfield data structure must start in position 1. (Normally the overlaying subfields will be defined using keyword OVERLAY(ds:1).) In the special case where the file contains only one record, the result data structure may be defined as in rule 1.
3. The result data structure can also be defined using LIKEDS(ds), where ds is an data structure following these rules.

If an input operation (CHAIN, READ, READC, READE, READP, READPE) does not retrieve a record because no record was found, because an error occurred in the operation, or because the last record was already retrieved (end of file), then no data is extracted and all fields in the program remain unchanged.

If you specify N as the operation extender of a CHAIN, READ, READE, READP, or READPE operation for an update disk file, a record is read without locking. If no operation extender is specified, the record is locked if the file is an update disk file.

To handle exceptions and errors that occur during file operations, you must specify an error indicator or a file error subroutine. Otherwise, exceptions and errors are handled by the default error handler.

Note: Input and output operations in subprocedures involving input and output specifications always use the global name, even if there is a local variable of the same name. For example, if the field name TOTALS is defined in the main source section, as well as in a subprocedure, any input or output operation in the subprocedure will use the field as defined in the main source section.

See “Database Null Value Support” on page 137 for information on handling files with null-capable fields.

Keys for File Operations

With the file operations CHAIN, DELETE, READE, READPE, SETGT and SETLL,the search argument, *search-arg*, must be the key or relative record number used to identify the record. For free-form calculations, a search argument may be:

1. A single field name
2. A klist name
3. A list of values, such as "(a:b:c+2)". Each part of the composite key may be any expression. Data types must match the corresponding key field, but lengths and data format do not have to match.
4. %KDS(ds{:*num*})

A composite key is formed from the subfields of the specified data structure in turn. Data types must match with the corresponding key field, but lengths and data format do not have to match. Rules for moving data from expression values to the key build area are the same as for operations code EVAL in that shorter search arguments are padded on the right with blanks and longer search arguments are truncated for type character. If *num* is specified, that is the number of subfields to use in the composite key.

For non-free-form calculations, only field names and klist names are allowed as search argument.

Operation extenders H, M, and R are allowed for CHAIN, DELETE, READE, READPE, SETGT, and SETLL when a list of search arguments or %KDS is specified. These extenders apply to the moving of the individual search argument to the search argument build area.

Indicator-Setting Operations

The indicator setting operation codes are:
- "SETOFF (Set Indicator Off)" on page 684
- "SETON (Set Indicator On)" on page 684

These operation codes are available only in the traditional syntax. In free-form syntax, you can set the value of *INxx to *ON or *OFF using the EVAL operation.

The following indicator-setting built-in function is available in both the traditional syntax and free-form syntax:
- "%NULLIND (Query or Set Null Indicator)" on page 460

The SETON and SETOFF operations set the indicators specified in positions 71 through 76 on and off, respectively. At least one resulting indicator must be specified in these positions.

See each operation for an explanation of its function.

Information Operations

The information operations are shown in the following table.

Table 44. Information Operations

Operation	Traditional Syntax	Free-Form Syntax
Get Time and Date	"TIME (Time of Day)" on page 707	• "%DATE (Convert to Date)" on page 422 • "%TIME (Convert to Time)" on page 487 • "%TIMESTAMP (Convert to Timestamp)" on page 488

The TIME operation allows the program to access the system time of day and system date at any time during program running. The date that is retrieved can be from your local system or from an iSeries server.

Initialization Operations

The initialization operations are:
- "CLEAR (Clear)" on page 539
- "RESET (Reset)" on page 668

The initialization operations provide run-time clearing and resetting of all elements in a window (entry field parts), structure (record format, data structure, array, or table) or a variable (field, subfield, or indicator).

These operations are available in both the traditional syntax and free-form syntax.

Memory Management Operations

The memory management operations are shown in the following table.

Table 45. Memory Management Operations

Operation	Traditional Syntax	Free-Form Syntax
Allocate Storage	"ALLOC (Allocate Storage)" on page 505	"%ALLOC (Allocate Storage)" on page 408
Free Storage	"DEALLOC (Free Storage)" on page 546	
Reallocate Storage	"REALLOC (Reallocate Storage with New Length)" on page 666	"%REALLOC (Reallocate Storage)" on page 464
Get the Address of a Variable	"%ADDR (Get Address of Variable)" on page 406	
Get the Address of a Procedure	"%PADDR (Get Procedure Address)" on page 463	

The ALLOC operation allocates heap storage and sets the result-field pointer to point to the storage. The storage is uninitialized.

The REALLOC operation changes the length of the heap storage pointed to by the result-field pointer. New storage is allocated and initialized to the value of the old storage. The data is truncated if the new size is smaller than the old size. If the new size is greater than the old size, the storage following the copied data is uninitialized. The old storage is released. The result-field pointer is set to point to the new storage.

The DEALLOC operation releases the heap storage that the result-field pointer is set to. If operational extender (N) is specified, the pointer is set to *NULL after a successful deallocation.

Storage is implicitly freed when the activation group ends. Setting LR on will not free any heap storage allocated by the module, but any pointers to heap storage will be lost

Misuse of heap storage can cause problems. The following example illustrates a scenario to avoid:

```
D Fld1         S                   25A    BASED(Ptr1)
D Fld2         S                    5A    BASED(Ptr2)
D Ptr1         S                     *
D Ptr2         S                     *
 ....
C                ALLOC     25              Ptr1
C                DEALLOC                   Ptr1
C* After this point, Fld1 should not be accessed since the
C* basing pointer Ptr1 no longer points to allocated storage.

C                CALL      'SOMEPGM'

C* During the previous call to 'SOMEPGM', several storage allocations
C* may have been done.  In any case, it is extremely dangerous to
C* make the following assignment, since 25 bytes of storage will
C* be filled with 'a'.  It is impossible to know what that storage
C* is currently being used for.
C                EVAL      Fld1 = *ALL'a'
```

Following are more problematic situations:

- A similar error can be made if a pointer is copied before being reallocated or deallocated. Great care must be taken when copying pointers to allocated storage, to ensure that they are not used after the storage is deallocated or reallocated.
- If a pointer to heap storage is copied, the copy can be used to deallocate or reallocate the storage. In this case, the original pointer should not be used until it is set to a new value.
- If a pointer to heap storage is passed as a parameter, the callee could deallocate or reallocate the storage. After the call returns, attempts to access the storage through pointer could cause problems.
- If a pointer to heap storage is set in the *INZSR, a later RESET of the pointer could cause the pointer to get set to storage that is no longer allocated.
- Another type of problem can be caused if a pointer to heap storage is lost (by being cleared, or set to a new pointer by an ALLOC operation, for example). Once the pointer is lost, the storage it pointed to cannot be freed. This storage is unavailable storage it pointed to cannot be freed. This storage is unavailable to be allocated since the system does not know that the storage is no longer addressable. The storage will not be freed until the activation group ends.

Message Operations

The message operation DSPLY displays a Message window. For more information, see “DSPLY (Display Message Window)” on page 562.

This operation is available in both the traditional syntax and free-form syntax.

Move Operations

The move operations are shown in the following table.

Table 46. Move Operations

Operation	Traditional Syntax	Free-Form Syntax
Move	“MOVE (Move)” on page 604	“EVALR (Evaluate expression, right adjust)” on page 573 or conversion built-in functions
Move an Array	“MOVEA (Move Array)” on page 619	(not allowed)

Table 46. Move Operations (continued)

Operation	Traditional Syntax	Free-Form Syntax
Move Left	"MOVEL (Move Left)" on page 626	"EVAL (Evaluate Expression)" on page 571 or conversionbuilt-in functions

Move operations transfer all or part of factor 2 to the result field. Factor 2 remains unchanged.

For a description of how data is moved, see each operation. For a description of how date-time data is moved when MOVE and MOVEL are used, see "Moving Date-Time Data" on page 370.

The source and target of the move operation can be of the same or different types, but some restrictions apply:

- For pointer moves, source and target must be the same type, either both basing pointers or both procedure pointers.
- When using MOVEA, both the source and target must be of the same type.
- MOVEA is not allowed for Date, Time or Timestamp fields.
- MOVE and MOVEL are not allowed for float fields or literals.

Resulting indicators can be specified only for character, graphic, UCS-2, and numeric result fields. For the MOVE and MOVEL operations, resulting indicators are not allowed if the result field is an unindexed array. For MOVEA, resulting indicators are not allowed if the result field is an array, regardless of whether or not it is indexed.

The P operation extender can only be specified if the result field is character, graphic, UCS-2, or numeric.

Moving Character, Graphic, UCS-2, and Numeric Data

When a character field is moved into a numeric result field, the digit portion of each character is converted to its corresponding numeric character and then moved to the result field. Blanks are transferred as zeros. For the MOVE operation, the zone portion of the rightmost character is converted to its corresponding sign and moved to the rightmost position of the numeric result field. It becomes the sign of the field. For the MOVEL operation, the zone portion of the rightmost character of factor 2 is converted and used as the sign of the result field (unless factor 2 is shorter than the result field) whether or not the rightmost character is included in the move operation.

If move operations are specified between numeric fields, the decimal positions specified for the factor 2 field are ignored. For example, if 1.00 is moved into a three-position numeric field with one decimal position, the result is 10.0.

Factor 2 may contain the figurative constants *ZEROS for moves to character or numeric fields. To achieve the same function for graphic fields, the user should code *ALLG'xx' (where 'xx' represents graphic zeros)

When moving data from a character source to graphic fields, if the source is a character literal, named constant, or *ALLm, it must be an even length and at least 2 bytes. When moving from a hexadecimal literal or *ALLx to a graphic field, the hexadecimal literal (or pattern) must be an even number of bytes.

When a character field is involved in a move from/to a graphic field, the source field must be an even length and at least 2 bytes.

When move operations are used to convert data from character to UCS-2 or from UCS-2 to character, the number of characters moved is variable since the character data may or may not contain graphic characters. For example, five UCS-2 characters can convert to:
- Five single-byte characters
- Five double-byte characters
- A combination of single-byte and double-byte characters

If the resulting data is too long to fit the result field, the data will be truncated. If the result is single-byte character, it is the responsibility of the user to ensure that the result contains complete characters.

If you specify operation extender P for a move operation, the result field is padded from the right for MOVEL and MOVEA, and from the left for MOVE. The pad characters are:
- Blank for character
- Double-byte blanks for graphic
- UCS-2 blanks for UCS-2
- 0 (zero) for numeric
- '0' for indicator

The padding takes place after the operation. If you use MOVE or MOVEL to move a field to an array, each element of the array will be padded. If you use these operations to move an array to an array and the result contains more elements than the factor 2 array, the same padding takes place but the extra elements are not affected. A MOVEA operation with an array name in the result field will pad the last element affected by the operation plus all subsequent elements.

When resulting indicators are specified for move operations, the result field determines which indicator is set on. If the result field is a character, graphic, or UCS-2 field, only the resulting indicator in positions 75 and 76 can be specified. This indicator is set on if the result field is all blanks. When the result field is numeric, all three resulting indicator positions may be used. These indicators are set on as follows:

High (71-72)
Set on if the result field is greater than 0.

Low (73-74)
Set on if the result field is less than 0.

Equal (75-76)
Set on if the result field is equal to 0.

Moving Date-Time Data

The MOVE and MOVEL operation codes can be used to move Date, Time and Timestamp data type fields.

The following combinations are allowed for the MOVE and MOVEL operation codes:
- Date to Date, Date to Timestamp, Date to Character or Numeric
- Time to Time, Time to Character or Numeric, Time to Timestamp
- Timestamp to Timestamp, Timestamp to Date, Timestamp to Time, Timestamp to Character or Numeric

- Character or Numeric to Date, Character or Numeric to Time, Character or Numeric to Timestamp

Factor 1 must be blank if both the source and the target of the move are Date, Time, or Timestamp fields. If Factor 1 is blank, the format of the Date, Time, or Timestamp field is used.

Otherwise, factor 1 contains the date or time format compatible with the character or numeric field that is the source or target of the operation. Any valid format may be specified. See "Date Data" on page 119, "Time Data" on page 135, and "Timestamp Data" on page 137.

Keep in mind the following when specifying factor 1:
- Time format *USA is not allowed for movement between Time and numeric fields.
- The formats *LONGJUL, *CYMD, *CMDY, and *CDMY are allowed in factor 1. (For more information see Table 15 on page 121.)
- A zero (0), specified at the end of a format (for example *MDY0), indicates that the character field does not contain separators.
- A two-digit year format (*MDY, *DMY, *YMD, and *JUL) can only represent dates in the range 1940 through 2039. A 3-digit year format (*CYMD, *CMDY, *CDMY) can only represent dates in the range 1900 through 2899. An error will be issued if conversion to a 2- or 3-digit year format is requested for dates outside this range.
- When MOVE and MOVEL are used to move character or numeric values to or from a timestamp, the character or numeric value is assumed to contain a timestamp.

Factor 2 is required and must be a character, numeric, Date, Time, or Timestamp value. It contains the field, array, array element, table name, literal, or named constant to be converted.

The following rules apply to factor 2:
- Separator characters must be valid for the specified format,
- If factor 2 is not a valid representation of a date or time or its format does not match the format specified in factor 1, an error is generated.
- If factor 2 contains UDATE or *DATE, factor 1 is optional and corresponds to the header specifications DATEDIT keyword
- If factor 2 contains UDATE and factor 1 entry is coded, it must be a date format with a two-digit year. If factor 2 contains *DATE and factor 1 is coded, it must be a date format with a 4-digit year.

The result field must be a Date, Time, Timestamp, numeric, or character variable. It can be a field, array, array element, or table name. The date or time is placed in the result field according to its defined format or the format code specified in factor 1. If the result field is numeric, separator characters will be removed, prior to the operation. The length is the length after removing the separator characters.

When moving from a Date to a Timestamp field, the time and microsecond portion of the timestamp are unaffected, however the entire timestamp is checked and an error message will be generated if it is not valid.

When moving from a Time to a Timestamp field, the microseconds part of the timestamp will be set to 000000. The date portion remains unaffected, but the entire timestamp will be checked and an error will be generated when it is not valid.

If character or numeric data is longer than required, only the leftmost data (rightmost for the MOVE operation) is used. Keep in mind that factor 1 determines the length of data to be moved. For example, if the format of factor 1 is *MDY for a MOVE operation from a numeric date, only the rightmost 6 digits of factor 2 would be used.

The P operation extender can only be specified if the result is character or numeric.

Examples of Converting a Character Field to a Date Field

Figure 124 on page 374 shows some examples of how to define and move 2- and 4-digit year dates between date fields, or between character and date fields.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 * Define two 8-byte character fields.
D CHR_8a          s              8a   inz('95/05/21')
D CHR_8b          s              8a   inz('abcdefgh')
 *
 * Define two 8-byte date fields.  To get a 2-digit year instead of
 * the default 4-digit year (for *ISO format), they are defined
 * with a 2-digit year date format, *YMD.  For D_8a, a separator (.)
 * is also specified.  Note that the format of the date literal
 * specified with the INZ keyword must be the same as the format
 * specified on the * control specification.  In this case, none
 * is specified, so it is the default, *ISO.
 *
D D_8a            s               d   datfmt(*ymd.)
D D_8b            s               d   inz(d'1995-07-31') datfmt(*ymd)
 *
 * Define a 10-byte date field.  By default, it has *ISO format.
D D_10            s               d   inz(d'1994-06-10')
 *
 *  D_10 now has the value 1995-05-21
 *
 *  Move the 8-character field to a 10-character date field D_10.
 *  It will contain the date that CHR_8a was initialized to, but
 *  with a 4-digit year and the format of D_10, namely,
 *  1995-05-21 (*ISO format).
 *
 *  Note that a format must be specified with built-in function
 *  %DATE to indicate the format of the character field.
 *
 /FREE
    D_10 = %DATE (CHR_8a: *YMD);
    //
    //  Move the 10-character date to an 8-character field CHR_8b.
    //  It will contain the date that was just moved to D_10, but with
    //  a 2-digit year and the default separator indicated by the *YMD
    //  format.
    //
    CHR_8b = %CHAR (D_10: *YMD);
    //
    //  Move the 10-character date to an 8-character date D_8a.
    //  It will contain the date that *  was just moved to D_10, but
    //  with a 2-digit year and a . separator since D_8a was defined
    //  with the (*YMD.) format.
    //
    D_8a = D_10;
    //
    //  Move the 8-character date to a 10-character date D_10
    //  It will contain the date that *  D_8b was initialized to,
    //  but with a 4-digit year, 1995-07-31.
    //
    D_10 = D_8b;
    //
    // After the last move, the fields will contain
    //  CHR_8b:  95/05/21
    //  D_8a:    95.05.21
    //  D_10:    1995-07-31
    //
    *INLR = *ON;
 /END-FREE
```

Figure 124. Moving character and date data

Result Operations

The following built-in functions work with the result of the previous operation:
- "%EQUAL (Return Exact Match Condition)" on page 440
- "%FOUND (Return Found Condition)" on page 444
- "%ERROR (Return Error Condition)" on page 441
- "%STATUS (Return File or Program Status)" on page 475

These built-in functions are available in both the traditional syntax and free-form syntax.

Size Operations

The following built-in functions return information about the size of a varible, field, constant, array, table, or data structure:
- "%DECPOS (Get Number of Decimal Positions)" on page 427
- "%LEN (Get or Set Length)" on page 452
- "%SIZE (Size of Constant or Field)" on page 472

These built-in functions are available in both the traditional syntax and free-form syntax.

String Operations

Table 47. String Operations

Operation	Traditional Syntax	Free-Form Syntax
Concatenate	"CAT (Concatenate Two Strings)" on page 526	+ operator
Check	"CHECK (Check Characters)" on page 533	"%CHECK (Check Characters)" on page 418
Check Reverse	"CHECKR (Check Reverse)" on page 536	"%CHECKR (Check Reverse)" on page 420
Create	"%STR (Get or Store Null-Terminated String)" on page 478	
Replace	"%REPLACE (Replace Character String)" on page 466	
Scan	"SCAN (Scan String)" on page 673	"%SCAN (Scan for Characters)" on page 468
Substring	"SUBST (Substring)" on page 696	"%SUBST (Get Substring)" on page 484
Translate	"XLATE (Translate)" on page 720	"%XLATE (Translate)" on page 498
Trim Blanks	"%TRIM (Trim Characters at Edges)" on page 490, "%TRIML (Trim Leading Characters)" on page 492, or "%TRIMR (Trim Trailing Characters)" on page 493	

The string operations include concatenation, scanning, substringing, translation, and verification. String operations can only be used on character, graphic, or UCS-2 fields.

Note:
- Strings are indexed from position 1.
- Figurative constants cannot be used in the factor 1, factor 2, or result fields.
- No overlapping in a data structure is allowed for factor 1 and the result field, or factor 2 and the result field.

Size Operations

When using string operations on graphic fields, all data in factor 1, factor 2, and result field must be graphic. When numeric values are specified for length, start position, and number of blanks for graphic characters, the values represent double-byte characters.

When using string operations on UCS-2 fields, all data in factor 1, factor 2, and the result field must be UCS-2. When numeric values are specified for length, start position, and number of blanks for UCS-2 characters, the values represent double-byte characters.

When using string operations on the graphic part of mixed-mode character data, the start position, length, and number of blanks represent single byte characters.

Note: Preserving data integrity is the user's responsibility.

Structured Programming Operations

The structured programming operations are shown in the following table.

Table 48. Structured Programming Operations

Operation	Traditional Syntax	Free-Form Syntax
And	"ANDxx (And)" on page 506	AND operator
Do	"DO (Do)" on page 554	"FOR (For)" on page 581
Do Until	"DOU (Do Until)" on page 556 or "DOUxx (Do Until)" on page 557	"DOU (Do Until)" on page 556
Do While	"DOW (Do While)" on page 559 or "DOWxx (Do While)" on page 560	"DOW (Do While)" on page 559
Else	"ELSE (Else)" on page 564	
Else If	"ELSEIF (Else If)" on page 565	
End	"ENDyy (End a Structured Group)" on page 566	
For	"FOR (For)" on page 581	
If	"IF (If)" on page 586 or "IFxx (If)" on page 587	"IF (If)" on page 586
Iterate	"ITER (Iterate)" on page 591	
Leave	"LEAVE (Leave a Do/For Group)" on page 596	
Or	"ORxx (Or)" on page 644	OR operator
Otherwise	"OTHER (Otherwise Select)" on page 645	
Select	"SELECT (Begin a Select Group)" on page 676	
When	"WHEN (When True Then Select)" on page 713 or "WHENxx (When True Then Select)" on page 714	"WHEN (When True Then Select)" on page 713

* **Restriction:** FOR and ENDFOR are unsupported in Java applications.

The rules for making the comparison on the ANDxx, DOUxx, DOWxx, IFxx, ORxx and WHENxx operation codes are the same as those given under "Compare Operations" on page 357.

In the ANDxx, DOUxx, DOWxx, IFxx, ORxx, and WHENxx operations, xx can be:

xx Meaning

GT Factor 1 is greater than factor 2.

LT Factor 1 is less than factor 2.

EQ Factor 1 is equal to factor 2.

NE Factor 1 is not equal to factor 2.

GE Factor 1 is greater than or equal to factor 2.

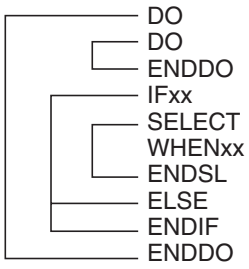
LE Factor 1 is less than or equal to factor 2.

In the ENDyy operation, yy can be:

yy	Meaning
CS	End for CASxx operation.
DO	End for DO, DOUxx, and DOWxx operation.
FOR	End for FOR operation.
IF	End for IFxx operation.
SL	End for SELECT operation.
Blanks	End for any structured operation.

Note: The yy in the ENDyy operation is optional.

If a structured group, in this case a do group, contains another complete structured group, together they form a nested structured group. Structured groups can be nested to a maximum depth of 100 levels. The following is an example of nested structured groups, three levels deep:



Remember the following when specifying structured groups:

- Each nested structured group must be completely contained within the outer level structured group.
- Each structured group must contain one of a DO, DOUxx, DOWxx, FOR, IFxx, or SELECT operation and its associated ENDyy operation.
- Branching into a structured group from outside the structured group may cause undesirable results.

Subroutine Operations

The subroutine operations are:
- "BEGACT (Begin Action Subroutine)" on page 508
- "ENDACT (End of Action Subroutine)" on page 568
- "BEGSR (Begin User Subroutine)" on page 511
- "ENDSR (End of User Subroutine)" on page 569
- "EXSR (Invoke User Subroutine)" on page 577
- "LEAVESR (Leave a Subroutine)" on page 598
- "CASxx (Conditionally Invoke Subroutine)" on page 524 (traditional syntax only)

All of these operations except CASxx are available in both the traditional syntax and free-form syntax.

A subroutine is a group of calculation specifications in a program that can be processed several times in that program.

Subroutine specifications must follow all other calculation operations that can be processed for a program; however, the PLIST, PARM, KLIST, KFLD, and DEFINE operations may be specified between an ENDSR operation (the end of one subroutine) and a BEGSR operation (the beginning of another subroutine) or after all subroutines. A subroutine can be called using an EXSR or CASxx operation anywhere in the calculation specifications. Subroutine lines can be identified by SR in positions 7 and 8. The only valid entries in positions 7 and 8 of a subroutine line are SR, AN, OR, or blanks.

For information on how to code a subroutine, see "Coding User Subroutines" on page 577.

Test Operations

The test operations are:
- "TEST (Test Date/Time/Timestamp)" on page 700
- "TESTB (Test Bit)" on page 703
- "TESTN (Test Numeric)" on page 705
- "TESTZ (Test Zone)" on page 706

The result of these operations is indicated by the resulting indicators.

TEST is available in both the traditional syntax and free-form syntax. The other operations are available only in the traditional syntax. See Figure 138 on page 413 for an example of how %BITAND can be used to duplicate the function of TESTB.

GUI Operations

The VisualAge RPG operations are:
- "BEGACT (Begin Action Subroutine)" on page 508
- "CLSWIN (Close Window)" on page 543
- "DSPLY (Display Message Window)" on page 562
- "ENDACT (End of Action Subroutine)" on page 568
- "GETATR (Retrieve Attribute)" on page 584
- "READS (Read Selected)" on page 666
- "SETATR (Set Attribute)" on page 678
- "SHOWWIN (Display Window)" on page 685
- "START (Start Component or Call Local Program)" on page 689

- "STOP (Stop Component)" on page 691

The VisualAge RPG operations work on either the user interface of the application (for example, SHOWWIN) or they work on components in the operation (for example, STOP). See each operation for an explanation of its function.

Qualified GUI Part Attribute Access

A qualified naming syntax is supported for accessing GUI part attributes in expressions or free-form calculations, as an alternative to the %GETATR and %SETATR builtin functions:

```
          <window-name>.<part-name>.<attribute-name>

eg.
     C                   eval      caltest.disparrows.checked =
     C                                caltest.calendar.montharrow
```

Notes:

1. %WINDOW and %PART are not valid in this format.
2. Attributes for the special part *Component are accessed as **component.*component.<attribute-name>*.
3. To support this qualified access to GUI part atttributes, the window names defined in the component's GUI definition are reserved as symbolic names (eg. field names) in the program, even within procedures.
4. This qualified attribute access does not affect the corresponding program fields for parts. To ensure that the attribute value and the value in the program field are the same, assign one to the other as needed.

   This applies only to attributes that have program fields mapped to them, such as the TEXT attribute for an entry field part.

   ```
   C                   EVAL      ENT0000B = INVENTORY.ENT0000B.TEXT
   ```

   Or, in the other direction:

   ```
    /FREE
         ENT0000B = *BLANKS;
         inventory.ent0000b.text = ENT0000B;
    /END-FREE
   ```

Chapter 24. Expressions

Expressions are a way to express program logic using free-form syntax. They can be used to write program statements in a more concise manner than fixed-form statements.

Expressions are simply groups of operands and operations, such as the following:

```
A+B*21
STRINGA + STRINGB
D = %ELEM(ARRAYNAME)
*IN01 OR (BALANCE > LIMIT)
SUM + TOTAL(ARRAY:%ELEM(ARRAY))
'The tax rate is ' + %editc(tax : 'A') + '%.'
```

Expressions may be coded in the following statements:

- "CALLP (Call a Prototyped Procedure or Program)" on page 522
- "CHAIN (Random Retrieval from a File)" on page 529 (free-form calculations only)
- "CLEAR (Clear)" on page 539(free-form calculations only)
- "DELETE (Delete Record)" on page 551 (free-form calculations only)
- "DSPLY (Display Message Window)" on page 562(free-form calculations only)
- "DOU (Do Until)" on page 556
- "DOW (Do While)" on page 559
- "EVAL (Evaluate Expression)" on page 571
- "EVALR (Evaluate expression, right adjust)" on page 573
- "FOR (For)" on page 581
- "IF (If)" on page 586
- "RETURN (Return to Caller)" on page 671
- "READE (Read Equal Key)" on page 658 (free-form calculations only)
- "READPE (Read Prior Equal)" on page 663 (free-form calculations only)
- "SETGT (Set Greater Than)" on page 679 (free-form calculations only)
- "SETLL (Set Lower Limit)" on page 681 (free-form calculations only)
- "SORTA (Sort an Array)" on page 686
- "WHEN (When True Then Select)" on page 713

Figure 125 on page 382 shows several examples of how expressions can be used:

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* The operations within the DOU group will iterate until the
* logical expression is true.  That is, either COUNTER is less
* than MAXITEMS or indicator 03 is on.
/FREE
    dou  counter < MAXITEMS or *in03;
    enddo;

    // The operations controlled by the IF operation will occur if
    // DUEDATE (a date variable) is an earlier date than
    // December 31, 1994.
    if DueDate < D'12-31-94';
    endif;

    // In this numeric expression, COUNTER is assigned the value
    // of COUNTER plus 1.
    Counter = Counter + 1;

    // This numeric expression uses a built-in function to assign the numb
    // of elements in the array ARRAY to the variable ARRAYSIZE.
    ArraySize = %elem (Array);

    // This expression calculates interest and performs half adjusting on
    // the result which is placed in the variable INTEREST.
    eval(h) Interest = Balance * Rate;

    // This character expression builds a sentence from a name and a
    // number using concatentation.  You can use built-in function
    // %CHAR, %EDITC, %EDITW or %EDITFLT to convert the numeric value
    // to character data.
    // This statement produces 'Id number for John Smith is 231 364'
    String = 'Id number for '
             + %trimr (First) + ' ' + %trimr (Last)
             + ' is ' + %editw (IdNum: '   &   ');

    // This expression adds a duration of 10 days to a date.
    DueDate = OriginalDate + %days(10);

    // This expression determines the difference in seconds between
    // two time values.
    Seconds = %diff (CompleteTime: t'09:00:00': *seconds);

    // This expression combines a date value and a time value into a
    // timestamp value.
    TimeStamp = TransactionDate + TransactionTime;
/END-FREE
```

Figure 125. Expression Examples

General Expression Rules

The following general rules apply to all expressions:

- Expressions are coded in the Extended-Factor 2 entry on the Calculation Specification or after the operation code on a free-form calculation.
- An expression can be continued on more than one specification. On a continuation specification, the only entries allowed are **C** in column 6 and the Extended-Factor 2 entry.

  No special continuation character is needed unless the expression is split within a literal or a name.
- Blanks (like parentheses) are required only to resolve ambiguity. However, they may be used to enhance readability.

Note that RPG will read as many characters as possible when parsing each token of an expression. For example,
  - X**DAY is X raised to the power of DAY
  - X* *DAY is X multiplied by *DAY
- The TRUNCNBR keyword on a control specification does not apply to calculations done within expressions. When overflow occurs during an expression operation, an exception is always issued.

Expression Operands

An operand can be any field name, named constant, literal, or prototyped procedure returning a value. In addition, the result of any operation can also be used as an operand to another operation. For example, in the expression A+B*21, the result of B*21 is an operand to the addition operation.

Expression Operators

Expression operators can be any of the following:

Unary Operations

Unary operations are coded by specifying the operator followed by one operand. The unary operators are:

+ The unary plus operation maintains the value of the numeric operand.

– The unary minus operation negates the value of the numeric operand.

NOT The logical negation operation returns '1' if the value of the indicator operand is '0' and '0' if the indicator operand is '1'. Note that the result of any comparison operation or operation **AND** or **OR** is a value of type indicator.

Binary Operations

Binary operations are coded by specifying the operator between the two operands. The binary operators are:

+ The meaning of this operation depends on the types of the operands. It can be used for:
- Adding two numeric values
- Adding a duration to a date, time, or timestamp.
- Concatenating two character, two graphic, or two UCS-2 values
- Adding a numeric offset to a basing pointer
- Combining a date and a time to yield a timestamp

– The meaning of this operation depends on the types of the operands. It can be used for:
- Subtracting two numeric values
- Subtracting a duration from a date, time, or timestamp.
- Subtracting a numeric offset from a basing pointer
- Subtracting two pointers

***** The multiplication operation is used to multiply two numeric values.

/ The division operation is used to divide two numeric values.

****** The exponentiation operation is used to raise a number to the power of another.

= The equality operation returns '1' if the two operands are equal, and '0' if not.

<> The inequality operation returns '0' if the two operands are equal, and '1' if not.

> The greater than operation returns '1' if the first operand is greater than the second.

>= The greater than or equal operation returns '1' if the first operand is greater or equal to the second.

< The less than operation returns '1' if the first operand is less than the second.

<= (less than or equal
The less than or equal operation returns '1' if the first operand is less or equal to the second.

AND The logical AND operation returns '1' if both operands have the value of indicator '1'.

OR The logical or operation returns '1' if either operand has the value of indicator '1'.

Built-in Functions
Built-in functions are discussed in "Built-In Functions (Alphabetically)" on page 405.

Assignment Operations

Assignment operations are coded by specifying the target of the assignment followed by an assignment operator followed by the expression to be assigned to the target. Assignment operators of the form op= (for example +=) use the target as one of the operands of the operation. The = assignment operator is used with the EVAL and EVALR operations. The op= assignment operators are used with the EVAL operation only. The assignment operators are:

- = The expression is assigned to the target
- += The expression is added to the target
- -= The expression is subtracted from the target
- *= The target is multiplied by the expression
- /= The target is divided by the expression
- **= The target is assigned the target raised to the power of the expression

User-Defined Functions
Any prototyped procedure that returns a value can be used within an expression. The call to the procedure can be placed anywhere that a value of the same type as the return value of the procedure would be used. For example, assume that procedure **MYFUNC** returns a character value. The following example shows three calls to **MYFUNC**:

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
     if  MyFunc (string1) = %trim (MyFunc (string2));
         %subst(X(3))= MyFunc('abc');
     endif;
/END-FREE
```

Figure 126. Using a Prototyped Procedure in an Expression

For more information on user-defined functions see Chapter 6, “Subprocedures and Prototypes,” on page 63.

Operation Precedence

The precedence of operations determines the order in which operations are performed within expressions. High precedence operations are performed before lower precedence operations.

Since parentheses have the highest precedence, operations within parentheses are always performed first.

Operations of the same precedence are evaluated in left to right order, except for **, which is evaluated from right to left.

Note that, although an expression is evaluated from left to right, this does not mean that the operands are also evaluated from left to right. See “Order of Evaluation” on page 397 for additional considerations.

This list indicates the precedence of operators from highest to lowest:
1. ()
2. built-in functions
3. unary +, unary –, NOT
4. **
5. *,/
6. binary +, binary –
7. =,>=,>,<=,<,<>
8. AND
9. OR

Figure 127 shows how precedence works.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* The following two operations produce different results although
* the order of operands and operators is the same.  Assume that
* PRICE = 100, DISCOUNT = 10, and TAXRATE = 0.15.
* The first EVAL would result in a TAX of 98.5.
* Since multiplication has a higher precedence than subtraction,
* DISCOUNT * TAXRATE is the first operation performed.  The result
* of that operation (1.5) is then subtracted from PRICE.

/FREE
    TAX = PRICE - DISCOUNT * TAXRATE;

    // The second EVAL would result in a TAX of 13.50.
    // Since parentheses have the highest precedence the operation
    // within parenthesis is performed first and the result of that
    // operation (90) is then multiplied by TAXRATE.

    TAX = (PRICE - DISCOUNT) * TAXRATE;
/END-FREE
```

Figure 127. Precedence Example

Data Types

All data types are allowed within expressions. However, specific operations only support certain data types as operands. For example, the * operation only allows numeric values as operands. Note that the relational and logical operations return a value of type indicator, which is a special type of character data. As a result, any relational or logical result can be used as an operand to any operation that expects character operands.

Data Types Supported by Expression Operands

The following tables summarize the data types supported by expression operands.

- Table 49 describes the type of operand allowed for each unary operator and the type of the result
- Table 50 describes the type of operands allowed for each binary operator and the type of the result
- Table 51 on page 387 describes the type of operands allowed for each built-in function and the type of the result. Prototyped procedures support whatever data types are defined in the prototype definition.

Table 49. Types Supported for Unary Operations

Operation	Operand Type	Result Type
– (negation)	Numeric	Numeric
+	Numeric	Numeric
NOT	Indicator	Indicator

Table 50. Operands Supported for Binary Operations

Operator	Operand 1 Type	Operand 2 Type	Result Type
+ (addition)	Numeric	Numeric	Numeric
+ (addition)	Date	Duration	Date
+ (addition)	Time	Duration	Time
+ (addition)	Timestamp	Duration	Timestamp
– (subtraction)	Numeric	Numeric	Numeric
- (subtraction)	Date	Duration	Date
- (subtraction)	Time	Duration	Time
- (subtraction)	Timestamp	Duration	Timestamp
* (multiplication)	Numeric	Numeric	Numeric
/ (division)	Numeric	Numeric	Numeric
** (exponentiation)	Numeric	Numeric	Numeric
+ (concatenation)	Character	Character	Character
+ (concatenation)	Graphic	Graphic	Graphic
+ (concatenation)	UCS-2	UCS-2	UCS-2
+ (add offset from pointer)	Basing Pointer	Numeric	Basing Pointer
- (subtract pointers)	Basing Pointer	Basing Pointer	Numeric
- (subtract offset from pointer)	Basing Pointer	Numeric	Basing Pointer
Note: For the following operations the operands may be of any type, but the two operands must be of the same type.			

Table 50. Operands Supported for Binary Operations (continued)

Operator	Operand 1 Type	Operand 2 Type	Result Type
= (equal to)	Any	Any	Indicator
>= (greater than or equal to)	Any	Any	Indicator
> (greater than)	Any	Any	Indicator
<= (less than or equal to)	Any	Any	Indicator
< (less than)	Any	Any	Indicator
<> (not equal to)	Any	Any	Indicator
AND (logical and)	Indicator	Indicator	Indicator
OR (logical or)	Indicator	Indicator	Indicator

Table 51. Types Supported for Built-in Functions

Operation	Operands	Result Type
%ABS	Numeric	Numeric
%ALLOC	Numeric	Pointer
%BITAND	Character:character{:character...}	Character
%BITAND	Numeric:numeric{:numeric...}	Numeric
%BITNOT	Character	Character
%BITNOT	Numeric	Numeric
%BITOR	Character:character{:character...}	Character
%BITOR	Numeric:numeric{:numeric...}	Numeric
%BITXOR	Character:character	Character
%BITXOR	Numeric:numeric	Numeric
%CHAR	Graphic, Numeric, UCS-2, Date, Time or Timestamp	Character
%CHECK	Character, Graphic, or UCS-2 {: Numeric}	Numeric
%CHECKR	Character, Graphic, or UCS-2 {: Numeric}	Numeric
%DATE	{Character, Numeric, or Timestamp {: Date Format}}	Date
%DAYS	Numeric	Numeric (duration)
%DEC	Character : Numeric constant : Numeric constant	Numeric (packed)
%DEC	Numeric> {: Numeric constant : Numeric constant}	Numeric (packed)
%DEC	Date, time or timestamp {: format}	Numeric (packed)
%DECH	Character : Numeric constant : Numeric constant	Numeric (packed)
%DECH	Numeric : Numeric constant : Numeric constant	Numeric (packed)
%DECPOS	Numeric	Numeric (unsigned)
%DIFF	Date, Time, or Timestamp : Date, Time, or Timestamp : Unit	Numeric (duration) (compatible with both)
%DIV	Numeric : Numeric	Numeric

Table 51. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
%EDITC	Non-float Numeric : Character Constant of Length 1 {: *CURSYM | *ASTFILL | character currency symbol}	Character (fixed length)
%EDITFLT	Numeric	Character (fixed length)
%EDITW	Non-float Numeric : Character Constant	Character (fixed length)
%EOF	{File name}	Indicator
%EQUAL	{File name}	Indicator
%ERROR		Indicator
%FLOAT	Character	Numeric (float)
%FLOAT	Numeric	Numeric (float)
%FOUND	{File name}	Indicator
%GETATR	Character : Character : Character	Any type except Pointer
%GRAPH	Character, Graphic, or UCS-2 {: ccsid}	Graphic
%HOURS	Numeric	Numeric (duration)
%INT	Character	Numeric (integer)
%INT	Numeric	Numeric (integer)
%INTH	Character	Numeric (integer)
%INTH	Numeric	Numeric (integer)
%LEN	Any	Numeric (unsigned)
%LOOKUPxx	Any : Any array {: Numeric {: Numeric}}	Numeric
%MINUTES	Numeric	Numeric (duration)
%MONTHS	Numeric	Numeric (duration)
%MSECONDS	Numeric	Numeric (duration)
%OCCUR	Multiple Occurrence Data Structure	Multiple Occurrence Data Structure
%OPEN	File name	Indicator
%REALLOC	Pointer : Numeric	Pointer
%REM	Numeric : Numeric	Numeric
%REPLACE	Character : Character {: Numeric {: Numeric}}	Character
%REPLACE	Graphic : Graphic {: Numeric {: Numeric}}	Graphic
%REPLACE	UCS-2 : UCS-2 {: Numeric {: Numeric}}	UCS-2
%SCAN	Character : Character {: Numeric}	Numeric (unsigned)
%SCAN	Graphic : Graphic {: Numeric}	Numeric (unsigned)
%SCAN	UCS-2 : UCS-2 {: Numeric}	Numeric (unsigned)
%SECONDS	Numeric	Numeric (duration)
%SQRT	Numeric	Numeric
%SETATR	Character : Character : Character	
%STATUS	{File name}	Numeric (zoned decimal)
%STR	Basing Pointer {: Numeric}	Character

Table 51. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
Note: When %STR appears on the left-hand side of an expression, the second operand is required.		
%SUBARR	Any: Numeric {:Numeric}	Any (same type as first operand)
%SUBDT	Date, Time, or Timestamp : Unit	Numeric
%SUBST	Character : Numeric {: Numeric}	Character
%SUBST	Graphic : Numeric {: Numeric}	Graphic
%SUBST	UCS-2 : Numeric {: Numeric}	UCS-2
%THIS		Object
%TIME	{Character, Numeric, or Timestamp {: Time Format}}	Time
%TIMESTAMP	{Character, Numeric, or Date {: Timestamp Format}}	Timestamp
%TLOOKUPxx	Any table: Any table {: Any}	Indicator
%TRIM	Character { : Character }	Character
%TRIM	Graphic { : Graphic}	Graphic
%TRIM	UCS-2 { : UCS-2 }	UCS-2
%TRIML	Character { : Character }	Character
%TRIML	Graphic { : Graphic}	Graphic
%TRIML	UCS-2 { : UCS-2 }	UCS-2
%TRIMR	Character { : Character }	Character
%TRIMR	Graphic { : Graphic}	Graphic
%TRIMR	UCS-2 { : UCS-2 }	UCS-2
%UCS2	Character or Graphic{:ccsid}	Varying length UCS-2 value
%UNS	Character	Numeric (unsigned)
%UNS	Numeric	Numeric (unsigned)
%UNSH	Character	Numeric (unsigned)
%UNSH	Numeric	Numeric (unsigned)
Note: For the following built-in functions, arguments must be literals, named constants or variables.		
%XFOOT	Numeric	Numeric
%XLATE	Character, Graphic, or UCS-2 : Character, Graphic, or UCS-2 : Character, Graphic, or UCS-2 {: Numeric}	Character, Graphic, or UCS-2
%YEARS	Numeric	Numeric (duration)
%PADDR	Character	Procedure or prototype pointer
%SIZE	Any {: *ALL}	Numeric (unsigned)
Note: For the following built-in functions, arguments must be variables. However, if an array index is specified, it may be any valid numeric expression.		
%ADDR	Any	Basing pointer
%ELEM	Any	Numeric (unsigned)

Table 51. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
%NULLIND	Any	Indicator
Note: The following built-in functions are not true built-in functions in that they do not return a value. They are used in some free-form file operations.		
%FIELDS	Any {: Any {: Any ...}	Not Applicable
%KDS	Data structure {: numeric }	Not Applicable

Format of Numeric Intermediate Results

For binary operations involving numeric fields, the format of the intermediate result depends on the format of the operands.

For the operators +, -, and *:

- If at least one operand has a float format, the result is float format.
- Otherwise, if at least one operand has packed-decimal, zoned-decimal, or binary format, the result has packed-decimal format.
- Otherwise, if at least one operand has integer format, the result has integer format.
- Otherwise, the result has unsigned format.
- For numeric literals that are not in float format:
  - If the literal is within the range of an unsigned integer, the literal is assumed to be an unsigned integer.
  - Otherwise, if the literal is within the range of an integer, the literal is assumed to be an integer.
  - Otherwise, the literal is assumed to be packed decimal.

For the / operator:

If one operand is float or the FLTDIV keyword is specified on the control specification, then the result of the / operator is float. Otherwise the result is packed-decimal.

For the ** operator:

The result is represented in float format.

Performance and 8-byte Arithmetic

By default, the compiler performs 4-byte arithmetic. 8-byte arithmetic only occurs if at least one operand is an 8-byte integer. From a performance perspective, 8-byte arithmetic is expensive and should be avoided.

Precision Rules for Numeric Operations

Unlike the fixed-form operation codes where you must always specify the result of each individual operation, RPG must determine the format and precision of the result of each operation within an expression.

If an operation has a result of format float, integer, or unsigned the precision is the maximum size for that format. Integer and unsigned operations produce 4-byte values and float operations produce 8-byte values.

However, if the operation has a packed-decimal, zoned decimal, or binary format, the precision of the result depends on the precisions of the operands.

It is important to be aware of the precision rules for decimal operations since even a relatively simple expression may have a result that may not be what you expect. For example, if the two operands of a multiplication are large enough, the result of the multiplication will have zero decimal places. If you are multiplying two 20 digit numbers, ideally you would need a 40 digit result to hold all possible results of the multiplication. However, since RPG supports numeric values only up to 31 digits, the result is adjusted to 31 digits. In this case, as many as 10 decimal digits are dropped from the result.

There are two sets of precision rules that you can use to control the sizes of intermediate values:

1. The default rules give you intermediate results that are as large as possible in order to minimize the possibility of numeric overflow. Unfortunately, in certain cases, this may yield results with zero decimal places if the result is very large.
2. The ″Result Decimal Positions″ precision rule works the same as the default rule except that if the statement involves an assignment to a numeric variable or a conversion to a specific decimal precision, the number of decimal positions of any intermediate result is never reduced below the desired result decimal places.

   In practice, you don't have to worry about the exact precisions if you examine the compile listing when coding numeric expressions. A diagnostic message indicates that decimal positions are being dropped in an intermediate result. If there is an assignment involved in the expression, you can ensure that the decimal positions are kept by using the ″Result Decimal Positions″ precision rule for the statement by coding operation code extender **(R)**.

   If the ″Result Decimal Position″ precision rule cannot be used (say, in a relational expression), built-in function **%DEC** can be used to convert the result of a sub-expression to a smaller precision which may prevent the decimal positions from being lost.

Using the Default Precision Rule

Using the default precision rule, the precision of a decimal intermediate in an expression is computed to minimize the possibility of numeric overflow. However, if the expression involves several operations on large decimal numbers, the intermediates may end up with zero decimal positions. (Especially, if the expression has two or more nested divisions.) This may not be what the programmer expects, especially in an assignment.

When determining the precision of a decimal intermediate, two steps occur:

1. The desired or ″natural″ precision of the result is computed.
2. If the natural precision is greater than 31 digits, the precision is adjusted to fit in 31 digits. This normally involves first reducing the number of decimal positions, and then if necessary, reducing the total number of digits of the intermediate.

This behavior is the default and can be specified for an entire module (using control specification keyword EXPROPTS(*MAXDIGITS) or for single free-form expressions (using operation code extender M).

Precision of Intermediate Results

Table 52 describes the default precision rules in more detail.

Table 52. Precision of Intermediate Results

Operation	Result Precision
Note: The following operations produce a numeric result. Ln is the length of the operand in digits where n is either r for result or a numeric representing the operand. Dn is the number of digits to the right of the decimal point where n is either r for result or a numeric representing the operand. T is the temporary value. Note that if any operand has a floating point representation (for example, it is the result of the exponentiation operator), the result also is a floating point value and the precision rules no longer apply. A floating point value has the precision available from double precision floating point representation.	
N1+N2	T=min (max (L1-D1, L2-D2)+1, 31) Dr=min (max (D1,D2), 31-t) Lr=t+Dr
N1-N2	T=min (max (L1-D1, L2-D2)+1, 31) Dr=min (max (D1,D2), 31-t) Lr=t+Dr
N1*N2	Lr=min (L1+L2, 31) Dr=min (D1+D2, 31-min ((L1-D1)+(L2-D2), 31))
N1/N2	Lr=31 Dr=max (31-((L1-D1)+D2), 0)
N1**N2	Double float
Note: The following operations produce a character result. Ln represents the length of the operand in number of characters.	
C1+C2	Lr=min(L1+L2,65535)
Note: The following operations produce a DBCS result. Ln represents the length of the operand in number of DBCS characters.	
D1+D2	Lr=min(L1+L2,16383)
Note: The following operations produce a result of type character with subtype indicator. The result is always an indicator value (1 character).	
V1=V2	1 (indicator)
V1>=V2	1 (indicator)
V1>V2	1 (indicator)
V1<=V2	1 (indicator)
V1<V2	1 (indicator)
V1<>V2	1 (indicator)
V1 AND V2	1 (indicator)
V1 OR V2	1 (indicator)

Example of Default Precision Rules

This example shows how the default precision rules work.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D FLD1            S              15P 4
D FLD2            S              15P 2
D FLD3            S               5P 2
D FLD4            S               9P 4
D FLD5            S               9P 4
CL0N01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C                   EVAL      FLD1 = FLD2/(((FLD3/100)*FLD4)+FLD5)
                                       (    1   )
                                      (         2    )
                                     (              3    )
                                  (     4                   )
```

Figure 128. Precision of Intermediate Results

When the above Calculation specification is processed, the resulting value assigned to FLD1 will have a precision of zero decimals, not the three decimals expected. The reason is that when it gets to the last evaluation (4 in the above example), the number to which the factor is scaled is negative. To see why, look at how the expression is evaluated.

1 Evaluate FLD3/100

Rules:

```
Lr = 31
Dr = max(31-((L1-D1)+D2),0)
   = max(31-((5-2)+0),0)
   = max(31-3,0)
   = 28
```

2 Evaluate (Result of 1 * FLD4)

Rules:

```
Lr = min(L1+L2,31)
   = min(31+9,31)
   = 31
Dr = min(D1+D2,31-min((L1-D1)+(L2-D2),31))
   = min(28+4,31-min((30-28)+(9-4),31))
   = min(32,31-min(4+5,31)
   = min(32,22)
   = 22
```

3 Evaluate (Result of 2 + FLD5)

Rules:

```
T  = min(max(L1-D1,L2-D2)+1,31)
   = min(max(31-22,9-4)+1,31)
   = min(max(9,5)+1,31)
   = min(10,31)
   = 10
Dr = min(max(D1,D2),31-T)
   = min(max(22,4),31-10)
   = min(22,21)
   = 21
Lr = T + Dr
   = 10 + 21 = 31
```

4 Evaluate FLD2/Result of 3

Rules:

```
Lr = 31
Dr = max(31-((L1-D1)+D2),0)
   = max(31-((15-2)+ 21),0)
   = max(31-(13+21),0)
   = max(-3,0)     ****  NEGATIVE NUMBER TO WHICH FACTOR IS SCALED   ****
   = 0
```

To avoid this problem, you can change the above expression so that the first evaluation is a multiplication rather than a division, that is, FLD3 * 0.01 or use the %DEC built-in function to set the sub-expression FLD3/100: %DEC(FLD3/100 : 15 : 4) or use operation extender (R) to ensure that the number of decimal positions never falls below 4.

Using the ″Result Decimal Position″ Precision Rules

The ″Result Decimal Position″ precision rule means that the precision of a decimal intermediate will be computed such that the number of decimal places will never be reduced smaller than the number of decimal positions of the result of the assignment. This is specified by:

1. **EXPROPTS(*RESDECPOS)** on the Control Specification. Use this to specify this behavior for an entire module.
2. Operation code extender **R** specified for a free-form operation.

Result Decimal Position rules apply in the following circumstances:

1. Result Decimal Position precision rules apply only to packed decimal intermediate results. This behavior does not apply to the intermediate results of operations that have integer, unsigned, or float results.
2. Result Decimal Position precision rules apply only where there is an assignment (either explicit or implicit) to a decimal target (packed, zoned, or binary).

   This can occur in the following situations:

   a. For an **EVAL** statement, the minimum decimal places is given by the decimal positions of the target of the assignment and applies to the expression on the right-hand side of the assignment. If half-adjust also applies to the statement, one extra digit is added to the minimum decimal positions (provided that the minimum is less than 31).
   b. For a **RETURN** statement, the minimum decimal places is given by the decimal positions of the return value defined on the **PI** specification for the procedure. If half-adjust also applies to the statement, one extra digit is added to the minimum decimal positions (provided that the minimum is less than 31).
   c. For a **VALUE** or **CONST** parameter, the minimum decimal positions is given by the decimal positions of the formal parameter (specified on the procedure prototype) and applies to the expression specified as the passed parameter.
   d. For built-in function **%DEC** and **%DECH** with explicit length and decimal positions specified, the minimum decimal positions is given by the third parameter of the built-in function and applies to the expression specified as the first parameter.

   The minimum number of decimal positions applies to the entire sub-expression unless overridden by another of the above operations. If half-adjust is specified (either as the **H** operation code extender, or by built-in function **%DECH**), the number of decimal positions of the intermediate result is never reduced below *N+1*, where *N* is the number of decimal positions of the result.

3. The Result Decimal Position rules do not normally apply to conditional expressions since there is no corresponding result. (If the comparisons must be performed to a particular precision, then **%DEC** or **%DECH** must be used on the two arguments.)

   On the other hand, if the conditional expression is embedded within an expression for which the minimum decimal positions are given (using one of the above techniques), then the Result Decimal Positions rules do apply.

Example of ″Result Decimal Position″ Precision Rules

The following examples illustrate the ″Result Decimal Position″ precision rules:

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* This example shows the precision of the intermediate values
* using the two precision rules.

D p1              s             13p 2
D p2              s             13p 2
D p3              s             13p 2
D p4              s             15p 9
D s1              s             13s 2
D s2              s             13s 2
D i1              s             10i 0
D f1              s              8f
D proc            pr             8p 3
D   parm1                       20p 5 value

 * In the following examples, for each sub-expression,
 * two precisions are shown.  First, the natural precision,
 * and then the adjusted precision.

 * Example 1:

 /FREE
    eval    p1 = p1 * p2 * p3;
    // p1*p2        -> P(26,4); P(26,4)
    // p1*p2*p3     -> P(39,6); P(31,0) (decimal positions are truncated)
     eval(r) p1 = p1 * p2 * p3;
    // p1*p2        -> P(26,4); P(26,4)
    // p1*p2*p3     -> P(39,6); P(31,2) (decimal positions do not drop
    //               below target decimal positions)
     eval(rh)p1 = p1 * p2 * p3;
    // p1*p2        -> P(26,4); P(26,5)
    // p1*p2*p3     -> P(39,6); P(31,3) (decimal positions do not drop
    //               below target decimals + 1)
     // Example 2:
     eval    p4 = p1 * p2 * proc (s1*s2*p4);
    // p1*p2        -> P(26,4); P(26,4)
    // s1*s2        -> P(26,4); P(26,4)
    // s1*s2*p4     -> P(41,13); P(31,3) (decimal positions are truncated)
    // p1*p2*proc() -> P(34,7); P(31,4)  (decimal positions are truncated)
     eval(r) p4 = p1 * p2 * proc (s1*s2*p4);
    // p1*p2        -> P(26,4); P(26,4)
    // s1*s2        -> P(26,4); P(26,4)
    // s1*s2*p4     -> P(41,13); P(31,5)
    // p1*p2*proc() -> P(34,7); P(31,7)  (we keep all decimals since we are
    //                already below target decimals)
 /END-FREE
```

Figure 129. Examples of Precision Rules

Short Circuit Evaluation

Relational operations AND and OR are evaluated from left to right. However, as soon as the value is known, evaluation of the expression stops and the value is returned. As a result, not all operands of the expression need to be evaluated.

For operation AND, if the first operand is false, then the second operand is not evaluated. Likewise, for operation OR, if the first operand is true, the second operand is not evaluated.

There are two implications of this behavior. First, an array index can be both tested and used within the same expression. The expression

```
I<=%ELEM(ARRAY) AND I>0 AND ARRAY(I)>10
```

will never result in an array indexing exception.

The second implication is that if the second operand is a call to a user-defined function, the function will not be called. This is important if the function changes the value of a parameter or a global variable.

Order of Evaluation

The order of evaluation of operands within an expression is not guaranteed. Therefore, if a variable is used twice anywhere within an expression, and there is the possibility of side effects, then the results may not be the expected ones.

For example, consider the source shown in Figure 130, where A is a variable, and FN is a procedure that modifies A. There are two occurrences of A in the expression portion of the second EVAL operation. ***If the left-hand side (operand 1) of the addition operation is evaluated first***, X is assigned the value 17, (5 + FN(5) = 5 + 12 = 17). ***If the right-hand side (operand 2) of the addition operation is evaluated first***, X is assigned the value 18, (6 + FN(5) = 6 + 12 = 18).

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *   A is a variable.  FN is procedure that modifies A.
 /free
     a = 5;
     x = a + fn(a);
 /end-free

P fn              B
D fn              PI             5P 0
D   parm     5P 0
 /free
     parm = parm + 1;
     return  2 * parm;
 /end-free
P fn              E
```

Figure 130. Sample coding of a call with side effects

Chapter 25. Built-In Functions

Built-in functions are similar to operation codes because they perform operations on data you specify. Built-in functions can be used in expressions. Additionally, constant-valued built-in functions can be used in named constants. These named constants can be used in any specification.

All built-in functions have the percent symbol (%) as their first character. The syntax of built-in functions is:

```
function-name{(argument{:argument...})}
```

Arguments for the function may be variables, constants, expressions, a prototyped procedure, or other built-in functions. An expression argument can include a built-in function. The following example illustrates this.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C*
C* This example shows a complex expression with multiple
C* nested built-in functions.
C*
C* %TRIM takes as its argument a string.  In this example, the
C* argument is the concatenation of string A and the string
C* returned by the %SUBST built-in function.  %SUBST will return
C* a substring of string B starting at position 11 and continuing
C* for the length returned by %SIZE minus 20.  %SIZE will return
C* the length of string B.
C*
C* If A is the string '     Toronto,' and B is the string
C* '  Ontario, Canada            ' then the argument for %TRIM will
C* be '     Toronto, Canada            ' and RES will have the value
C* 'Toronto, Canada'.
C*
C                   EVAL      RES = %TRIM(A + %SUBST(B:11:%SIZE(B) - 20))
```

Figure 131. Built-in Function Arguments Example

See the individual built-in function descriptions for details on the arguments that are allowed.

Unlike operation codes, built-in functions return a value rather than placing a value in a result field. The following example illustrates this difference.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* In the following example, CITY contains the string
C* 'Toronto, Ontario'.  The SCAN operation is used to locate the
C* separating blank, position 9 in this illustration.  SUBST
C* places the string 'Ontario' in field TCNTRE.
C*
C* Next, TCNTRE is compared to the literal 'Ontario' and
C* 1 is added to CITYCNT.
C*
C     ' '           SCAN      CITY          C
C                   ADD       1             C
C                   SUBST     CITY:C        TCNTRE
C    'Ontario'      IFEQ      TCNTRE
C                   ADD       1             CITYCNT
C                   ENDIF
C*
C* In this example, CITY contains the same value, but the
C* variable TCNTRE is not necessary since the %SUBST built-in
C* function returns the appropriate value.  In addition, the
C* intermediary step of adding 1 to C is simplified since
C* %SUBST accepts expressions as arguments.
C*
C     ' '           SCAN      CITY          C
C                   IF        %SUBST(CITY:C+1) = 'Ontario'
C                   EVAL      CITYCNT = CITYCNT+1
C                   ENDIF
```

Figure 132. Built-in Function Example

Note that the arguments used in this example (the variable CITY and the expression C+1) are analogous to the factor values for the SUBST operation. The return value of the function itself is analogous to the result. In general, the arguments of the built-in function are similar to the factor 1 and factor 2 fields of an operation code.

Another useful feature of built-in functions is that they can simplify maintenance of your code when used on the definition specification. The following example demonstrates this feature.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D*
D* In this example, CUSTNAME is a field in the
D* externally described data structure CUSTOMER.
D* If the length of CUSTNAME is changed, the attributes of
D* both TEMPNAME and NAMEARRAY would be changed merely by
D* recompiling.  The use of the %SIZE built-in function means
D* no changes to your code would be necessary.
D*
D CUSTOMER      E DS
D                 DS
D TEMPNAME                              LIKE(CUSTNAME)
D NAMEARRAY                         1   OVERLAY(TEMPNAME)
D                                       DIM(%SIZE(TEMPNAME))
```

Figure 133. Simplified Maintenance with Built-in Functions

Built-in functions can be used in expressions on the extended-factor 2 calculation specification and with keywords on the definition specification. When used with definition specification keywords, the value of the built-in function must be known at compile time and the argument cannot be an expression.

The following table lists the built-in functions, their arguments, and the value they return.

Name	Arguments	Value Returned
%ABS	numeric expression	absolute value of expression
%ADDR	variable name	address of variable
%ALLOC	number of bytes to allocate	pointer to allocated storage
%BITAND	character, numeric	bit wise ANDing of the bits of all the arguments
%BITNOT	character, numeric	bit-wise reverse of the bits of the argument
%BITOR	character, numeric	bit-wise ORing of the bits of all the arguments
%BITXOR	character, numeric	bit-wise exclusive ORing of the bits of the two arguments
%CHAR	graphic, UCS-2, numeric, date, time, or timestamp expression {: date, time, or timestamp format}	value in character format
%CHECK	comparator string:string to be checked{:start position}	first position of a character that is not in the comparator string, or zero if not found
%CHECKR	comparator string:string to be checked{:start position}	last position of a character that is not in the comparator string, or zero if not found
%DATE	{value {: date format}}	the date that corresponds to the specified *value*, or the current system date if none is specified
%DAYS	number of days	number of days as a duration
%DEC	numeric expression {:digits:decpos} character expression: digits:decpos date, time or timestamp expression	value in packed numeric format {:format}
%DECH	numeric or character expression: digits:decpos	half-adjusted value in packed numeric format
%DECPOS	numeric expression	number of decimal digits
%DIFF	date or time expression: date or time expression: unit	difference between the two dates, times, or timestamps in the specified unit
%DIV	dividend: divisor	the quotient from the division of the two arguments
%EDITC	not-float numeric expression:edit code{:*CURSYM | *ASTFILL | currency symbol}	string representing edited value
%EDITFLT	numeric expression	character external display representation of float
%EDITW	non-float numeric expression:edit word	string representing edited value
%ELEM	array, table, or multiple occurrence data structure name	number of elements or occurrences
%EOF	{file name}	'1' if the most recent cycle input, read operation, or write to a subfile (for a particular file, if specified) ended in an end-of-file or beginning-of-file condition; and, when a file is specified, if a more recent OPEN, CHAIN, SETGT or SETLL to the file was not successful
		'0' otherwise

Name	Arguments	Value Returned
%EQUAL	{file name}	'1' if the most recent SETLL (for a particular file, if specified) or LOOKUP operation found an exact match
		'0' otherwise
%ERROR		'1' if the most recent operation code with extender 'E' specified resulted in an error
		'0' otherwise
%FIELDS	list of fields to be updated	not applicable
%FLOAT	numeric or character expression	value in float form
%FOUND	{file name}	'1' if the most recent relevant operation (for a particular file, if specified) found a record (CHAIN, DELETE, SETGT, SETLL), an element (LOOKUP), or a match (CHECK, CHECKR and SCAN)
		'0' otherwise
%GETATR	window name, part name, attribute name, %PART, %WINDOW	value of attribute
%GRAPH	character, graphic, or UCS-2 expression	value in graphic format
%HOURS	number of hours	number of hours as a duration
%INT	numeric or character expression	value in integer format
%INTH	numeric or character expression	half-adjusted value in integer format
%KDS	data structure containing keys {: number of keys}	not applicable
%LEN	any expression	length in digits or characters
%LOOKUPxx	argument: array{:start index {:number of elements}}	array index of the matching element
%MINUTES	number of minutes	number of minutes as a duration
%MONTHS	number of months	number of months as a duration
%MSECONDS	number of microseconds	number of microseconds as a duration
%NULLIND	null-capable fieldname	value in indicator format representing the null indicator setting for the null-capable field
%OCCUR	multiple-occurrence data structure name	current occurrence of the multiple-occurrence data structure
%OPEN	file name	'1' if the specified file is open
		'0' if the specified file is closed
%PADDR	procedure or prototype name	address of procedure or prototype
%REALLOC	pointer: numeric expression	pointer to allocated storage
%REM	dividend: divisor	the remainder from the division of the two arguments
%REPLACE	replacement string: source string{:start position {:source length to replace}}	string produced by inserting replacement string into source string, starting at start position and replacing the specified number of characters
%SCAN	search argument:string to be searched{:start position}	first position of search argument in string or zero if not found
%SECONDS	number of seconds	number of seconds as a duration

Name	Arguments	Value Returned
%SETATR	window name, part name, attribute name, %PART, %WINDOW	none
%SIZE	variable, array, or literal{:*ALL}	size of variable or literal
%SQRT	numeric value	square root of the numeric value
%STATUS	{file name}	0 if no program or file error occurred since the most recent operation code with extender 'E' specified
		most recent value set for any program or file status, if an error occurred
		if a file is specified, the value returned is the most recent status for that file
%STR	pointer{:maximum length}	characters addressed by pointer argumentup to but not including the first x'00'
%SUBARR	array name:start index{:number of elements}	array subset
%SUBDT	date or time expression: unit	an unsigned numeric value that contains the specified portion of the date or time value
%SUBST	string:start{:length}	substring
%THIS		the class instance for the native method
%TIME	{value {: time format}}	the time that corresponds to the specified *value*, or the current system time if none is specified
%TIMESTAMP	{(value {: timestamp format})}	the timestamp that corresponds to the specified *value*, or the current system timestamp if none is specified
%TLOOKUPxx	argument: search table {: alternate table}	'*ON' if there is a match
		'*OFF' otherwise
%TRIM	string {: characters to trim}	string with left and right blanks or specified characters trimmed
%TRIML	string {: characters to trim}	string with left blanks or specified characters trimmed
%TRIMR	string {: characters to trim}	string with right blanks or specified characters trimmed
%UCS2	character or graphic expression	value in UCS-2 format
%UNS	numeric or character expression	value in unsigned format
%UNSH	numeric or character expression	half-adjusted value in unsigned format
%XFOOT	array expression	sum of the elements
%XLATE	from-characters: to-characters: string {: start position}	the string with from-characters replaced by to-characters
%YEARS	number of years	number of years as a duration

For more information on using built-in functions, see:
- Chapter 18, "Definition Specifications," on page 255
- "Extended Factor 2 Syntax" on page 316
- Chapter 24, "Expressions," on page 381
- "DOU (Do Until)" on page 556
- "DOW (Do While)" on page 559
- "EVAL (Evaluate Expression)" on page 571
- "IF (If)" on page 586
- "RETURN (Return to Caller)" on page 671

- "WHEN (When True Then Select)" on page 713

Built-In Functions (Alphabetically)

The following sections describe the built-in functions.

%ABS (Absolute Value of Expression)

%ABS

%ABS returns the absolute value of the numeric expression specified as parameter. If the value of the numeric expression is non-negative, the value is returned unchanged. If the value is negative, the value returned is the value of the expression but with the negative sign removed.

%ABS may be used either in expressions or as parameters to keywords. When used with keywords, the operand must be a numeric literal, a constant name representing a numeric value, or a built-in function with a numeric value known at compile-time.

For more information, see "Arithmetic Operations" on page 348 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name +++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D f8         s              8f   inz (-1)
D i10        s             10i 0 inz (-123)
D p7         s              7p 3 inz (-1234.567)

 /FREE
    f8  = %abs (f8);        // "f8" is now 1.
    i10 = %abs (i10 - 321); // "i10" is now 444.
    p7  = %abs (p7);        // "p7" is now 1234.567.
 /END-FREE
```

Figure 134. %ABS Example

%ADDR (Get Address of Variable)

```
%ADDR(variable)
%ADDR(variable(index))
%ADDR(variable(expression))
```

%ADDR returns a value of type basing pointer. The value is the address of the specified variable. It may only be compared with and assigned to items of type basing pointer.

If %ADDR with an array index parameter is specified as a parameter for the definition specification keywords INZ or CONST, the array index must be known at compile-time. The index must be either a numeric literal or a numeric constant.

In an EVAL operation where the result of the assignment is an array with no index, %ADDR on the right hand side of the assignment operator has a different meaning depending on the argument for the %ADDR. If the argument for %ADDR is an array name without an index and the result is an array name, each element of the result array contains the address of the beginning of the argument array. If the argument for %ADDR is an array name with an index of (*), then each element of the result array will contain the address of the corresponding element in the argument array. This is illustrated in Figure 135 on page 407.

If the variable specified as parameter is a table, multiple occurrence data structure, or subfield of a multiple occurrence data structure, the address is the address of the current table index or occurrence number.

If the variable is based, %ADDR returns the value of the basing pointer for the variable. If the variable is a subfield of a based data structure, the value of %ADDR is the value of the basing pointer plus the offset of the subfield.

If the variable is specified as a PARM of the *ENTRY PLIST, %ADDR returns the address passed to the program by the caller.

When the argument of %ADDR cannot be modified, %ADDR can only be used in a comparison operation. An example of an argument that cannot be modified is a read-only reference parameter (CONST keyword specified on the Procedure Interface).

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
 *
 * The following set of definitions is valid since the array
 * index has a compile-time value
 *
D   ARRAY         S             20A   DIM (100)
 * Set the pointer to the address of the seventh element of the array.
D   PTR           S               *   INZ (%ADDR(ARRAY(SEVEN)))
D   SEVEN         C                   CONST (7)
 *
D DS1             DS                  OCCURS (100)
D                                 20A
D   SUBF                          10A
D                                 30A
D CHAR10          S               10A   BASED (P)
D PARRAY          S                 *   DIM(100)


 /FREE
    %OCCUR(DS1) = 23;
    SUBF = *ALL'abcd';
    P = %ADDR (SUBF);
    IF CHAR10 = SUBF;
       // This condition is true.
    ENDIF;
    IF %ADDR (CHAR10) = %ADDR (SUBF);
       // This condition is also true.
    ENDIF;
    // The following statement also changes the value of SUBF.
    CHAR10 = *ALL'efgh';
    IF CHAR10 = SUBF;
       // This condition is still true.
    ENDIF;
    //---------------------------------------------------------------
    %OCCUR(DS1) = 24;
    IF CHAR10 = SUBF;
      // This condition is no longer true.
    ENDIF;
    //---------------------------------------------------------------
    // The address of an array element is taken using an expression
    // as the array index.
    P = %ADDR (ARRAY (X + 10));
    //---------------------------------------------------------------
    // Each element of the array PARRAY contains the address of the
    // first element of the array ARRAY.
    PARRAY = %ADDR(ARRAY);
    // Each element of the array PARRAY contains the address of the
    // corresponding element of the array ARRAY.
    PARRAY = %ADDR(ARRAY(*));

    // The first three elements of the array PARRAY
    // contain the addresses of the first three elements
    // of the array ARRAY.
    %SUBARR(PARRAY : 1 : 3) = %ADDR(ARRAY(*));
 /END-FREE
```

Figure 135. %ADDR Example

%ALLOC (Allocate Storage)

```
%ALLOC(num)
```

%ALLOC returns a pointer to newly allocated heap storage of the length specified. The newly allocated storage is uninitialized.

The parameter must be a non-float numeric value with zero decimal places. The length specified must be between 1 and 16776704.

For more information, see "Memory Management Operations" on page 367.

If the operation cannot complete successfully, exception 00425 or 00426 is issued.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
  // Allocate an area of 200 bytes
  pointer = %ALLOC(200);
/END-FREE
```

Figure 136. %ALLOC Example

%BITAND (Bitwise AND Operation)

```
%BITAND(expr:expr{:expr...})
```

%BITAND returns the bit-wise ANDing of the bits of all the arguments. That is, the result bit is ON when all of the corresponding bits in the arguments are ON, and OFF otherwise.

The arguments to this built-in function can be either character or numeric. For numeric arguments, if they are not integer or unsigned, they are first converted to integer. If the value does not fit in an 8-byte integer, a numeric overflow exception is issued.

%BITAND can have two or more arguments. All arguments must be the same type, either character or numeric. The result type is the same as the types of the arguments. For numeric arguments, the result is unsigned if all arguments are unsigned, and integer otherwise.

The length is the length of the largest operand. If the arguments have different lengths, they are padded on the left with bit zeros for numeric arguments. Shorter character arguments are padded on the right with bit ones.

%BITAND can be coded in any expression. It can also be coded as the argument to a File or Definition Specification keyword if all arguments are known at compile-time. If all arguments of this built-in function are hex literals, the compiler produces a constant-folded result that is a hex literal.

Please see Figure 137 on page 413, Figure 138 on page 413, and Figure 139 on page 414 for examples demonstrating the use of %BITAND.

For more information, see “Bit Operations” on page 352 or Chapter 25, “Built-In Functions,” on page 399.

%BITNOT (Invert Bits)

```
%BITNOT(expr)
```

%BITNOT returns the bit-wise inverse of the bits of the argument. That is, the result bit is ON when the corresponding bit in the argument is OFF, and OFF otherwise.

The argument to this built-in function can be either character or numeric. For numeric arguments, if they are not integer or unsigned, they are first converted to integer. If the value does not fit in an 8-byte integer, a numeric overflow exception is issued.

%BITNOT takes just one argument. The result type is the same as the types of the arguments. For numeric arguments, the result is unsigned if all arguments are unsigned, and integer otherwise.

The length is the length of the largest operand. If the arguments have different lengths, they are padded on the left with bit zeros for numeric arguments.

%BITNOT can be coded in any expression. It can also be coded as the argument to a File or Definition Specification keyword if all arguments are known at compile-time. If all arguments of this built-in function are hex literals, the compiler produces a constant-folded result that is a hex literal.

Please see Figure 137 on page 413 for an example demonstrating the use of %BITNOT.

For more information, see “Bit Operations” on page 352 or Chapter 25, “Built-In Functions,” on page 399.

%BITOR (Bitwise OR Operation)

```
%BITOR(expr:expr{:expr...})
```

%BITOR returns the bit-wise ORing of the bits of all the arguments. That is, the result bit is ON when any of the corresponding bits in the arguments are ON, and OFF otherwise.

The arguments to this built-in function can be either character or numeric. For numeric arguments, if they are not integer or unsigned, they are first converted to integer. If the value does not fit in an 8-byte integer, a numeric overflow exception is issued.

%BITOR can have two or more arguments. All arguments must be the same type, either character or numeric. However, when coded as keyword parameters, these two BIFs can have only two arguments. The result type is the same as the types of the arguments. For numeric arguments, the result is unsigned if all arguments are unsigned, and integer otherwise.

The length is the length of the largest operand. If the arguments have different lengths, they are padded on the left with bit zeros for numeric arguments. Shorter character arguments are padded on the right with bit zeros.

%BITOR can be coded in any expression. It can also be coded as the argument to a File or Definition Specification keyword if all arguments are known at compile-time. If all arguments of this built-in function are hex literals, the compiler produces a constant-folded result that is a hex literal.

Please see Figure 137 on page 413 for an example demonstrating the use of %BITOR.

For more information, see “Bit Operations” on page 352 or Chapter 25, “Built-In Functions,” on page 399.

%BITXOR (Bitwise Exclusive-OR Operation)

```
%BITXOR(expr:expr)
```

%BITXOR returns the bit-wise exclusive ORing of the bits of the two arguments. That is, the result bit is ON when just one of the corresponding bits in the arguments are ON, and OFF otherwise.

The argument to this built-in function can be either character or numeric. For numeric arguments, if they are not integer or unsigned, they are first converted to integer. If the value does not fit in an 8-byte integer, a numeric overflow exception is issued.

%BITXOR takes exactly two arguments. The result type is the same as the types of the arguments. For numeric arguments, the result is unsigned if all arguments are unsigned, and integer otherwise.

The length is the length of the largest operand. If the arguments have different lengths, they are padded on the left with bit zeros for numeric arguments. Shorter character arguments are padded on the right with bit zeros .

%BITXOR can be coded in any expression. It can also be coded as the argument to a File or Definition Specification keyword if all arguments are known at compile-time. If all arguments of this built-in function are hex literals, the compiler produces a constant-folded result that is a hex literal.

For more information, see “Bit Operations” on page 352 or Chapter 25, “Built-In Functions,” on page 399.

Examples of Bit Operations

```
D const           c                   x'0007'

D ch1             s              4a   inz(%BITNOT(const))
 * ch1 is initialized to x'FFF82020'
D num1            s              5i 0 inz(%BITXOR(const:x'000F'))
 * num is initialized to x'0008', or 8

D char2a          s              2a
D char2b          s              2a
D uA              s              5u 0
D uB              s              3u 0
D uC              s              5u 0
D uD              s              5u 0

C                   eval      char2a = x'FE51'
C                   eval      char2b = %BITAND(char10a : x'0F0F')
 * operand1   = b'1111 1110 0101 0001'
 * operand2   = b'0000 1111 0000 1111'
 * bitwise AND:   0000 1110 0000 0001
 * char2b = x'0E01'
C                   eval      uA = x'0123'
C                   eval      uB = x'AB'
C                   eval      uc = x'8816'
C                   eval      uD = %BITOR(uA : uB : uC)
 * operand1   = b'0000 0001 0010 0011'
 * operand2   = b'0000 0000 1010 1011'   (fill with x'00')
 * operand3   = b'1000 1000 0001 0110'
 * bitwise OR:    1000 1001 1011 1111
 * uD = x'89BF'
```

Figure 137. Using Bit Operations

```
 * This example shows how to duplicate the function of TESTB using %BITAND
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D fld1            s              1a
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C                   testb     x'F1'         fld1                 010203
    * Testing bits     1111 0001
    * If FLD1 = x'00' (0000 0000), the indicators have the values '1' '0' '0'
    *   (all tested bits are off)
    * If FLD1 = x'15' (0001 0101), the indicators have the values '0' '1' '0'
    *   (some tested bits are off and some are on)
    * If FLD1 = x'F1' (1111 0001), the indicators have the values '0' '0' '1'
    *   (all tested bits are on)
 /free
       // this code performs the equivalent of the TESTB operation above

       // test if all the "1" bits in x'F1' are off in FLD1
       *in01 = %bitand(fld1 : x'F1') = x'00';

       // test if some of the "1" bits in x'F1' are on
       // and some are off in FLD1
       *in02 = %bitand(fld1 : x'F1') <> x'00'
          and  %bitand(fld1 : x'F1') <> x'F1';

       // test if all the "1" bits in x'F1' are on in FLD1
       *in03 = %bitand(fld1 : x'F1') = x'F1';
 /end-free
```

Figure 138. Deriving TESTB Functionality from %BITAND

```
* This example shows how to duplicate the function of
* BITON and BITOFF using %BITAND, %BITNOT, and %BITOR
  DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
  D fld1            s              1a   inz(x'01')
  D fld2            s              1a   inz(x'FF')
  CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
  C                   biton     x'F4'         fld1
   * fld1 has an initial value of x'01' (0000 0001)
   * The 1 bits in x'F4' (1111 0100) are set on
   * fld1 has a final value of x'F5' (1111 0101)
  C                   bitoff    x'F1'         fld2
   * fld2 has an initial value of x'FF' (1111 1111)
   * The 1 bits in x'F1' (1111 0001) are set off
   * fld2 has a final value of x'0E' (0000 1110)
   /free
          // this code performs the equivalent of the
          // BITON and BITOFF operations above
          // Set on the "1" bits of x'F4' in FLD1
          fld1 = %bitor(fld1 : x'F4');
          // Set off the "1" bits of x'F1' in FLD2
          fld2 = %bitand(fld2 : %bitnot(x'F1'));

          /end-free
```

Figure 139. BITON/BITOFF Functionality Using Built In Functions

```
D c1              s              2a   inz(x'ABCD')
D c2hh            s              2a   inz(x'EF12')
D c2hl            s              2a   inz(x'EF12')
D c2lh            s              2a   inz(x'EF12')
D c2ll            s              2a   inz(x'EF12')
 /free
        // mhhzo     c1            c2hh
        // c2hh becomes x'AF12'
        %subst(c2hh:1:1)
               = %bitor(%bitand(x'0F'
                                : %subst(c2hh:1:1))
                        : %bitand(x'F0'
                                : %subst(c1:1:1)));
        // c2hl becomes x'EFA2'
        // mhlzo     c1            c2hl
        %subst(c2hl:%len(c2hl):1)
               = %bitor(%bitand(x'0F'
                                : %subst(c2hl:%len(c2hl):1))
                        : %bitand(x'F0'
                                : %subst(c1:1:1)));
        // mlhzo     c1            c2lh
        // c2lh becomes x'CF12'
        %subst(c2lh:1:1)
               = %bitor(%bitand(x'0F'
                                : %subst(c2lh:1:1))
                        : %bitand(x'F0'
                                : %subst(c1:%len(c1):1)));
        // mhllo     c1            c2ll
        // c2ll becomes x'EFC2'
        %subst(c2ll:%len(c2hl):1)
               = %bitor(%bitand(x'0F'
                                : %subst(c2ll:%len(c2ll):1))
                        : %bitand(x'F0'
                                : %subst(c1:%len(c1):1)));
```

Figure 140. Deriving MxxZO functionality from %BITOR and %BITAND

%CHAR (Convert to Character Data)

```
%CHAR(expression{:format})
```

%CHAR converts the value of the expression from graphic, UCS-2, numeric, date, time or timestamp data to type character. The converted value remains unchanged, but is returned in a format that is compatible with character data.

If the parameter is a constant, the conversion will be done at compile time.

If a UCS-2 conversion results in substitution characters, a warning message will be given in the compiler listing if the parameter is a constant. Otherwise, status 00050 will be set at run time but no error message will be given.

For graphic data, the value returned is two bytes for each graphic field. For example, if a 5 character graphic field is converted, the returned value is 10 characters (10 bytes of graphic data). If the value of the expression has a variable length, the value returned is in varying format.

For date, time, or timestamp data, the second parameter contains the date, time, or timestamp format to which the returned character data is converted. The value returned will include separator characters unless the format specified is followed by a zero.

For numeric data, if the value of the expression is float, the result will be in float format (for example '+1.125000000000000E+020'). Otherwise, the result will be in decimal format with a leading negative sign if the value is negative, and without leading zeros. The character used for any decimal point will be the character indicated by the control specification DECEDIT keyword (default is '.'). For example, %CHAR of a packed(7,3) expression might return the value '-1.234'.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D Name            S             20G   VARYING INZ(G'XXYYZZ')
D date            S               D   INZ(D'1997/02/03')
D time            S               T   INZ(T'12:23:34')
D result          S            100A   VARYING
D points          S             10i 0 INZ(234)

 *---------------------------------------------------------------
 * To format the time and date with the default formats, use this:
 *---------------------------------------------------------------
 /FREE
   result = 'It is ' + %CHAR(time) + ' on ' + %CHAR(date);
   // If the default formats are both *USA,
   // result = 'It is 12:23 PM on 02/03/1997'

   //---------------------------------------------------------------
   // To format the time and date with specific formats, use this:
   //---------------------------------------------------------------
   result = 'It is ' + %CHAR(time : *hms:)
            + ' on ' + %CHAR(date : *iso);
   // result = 'It is 12:23:34 on 1997-02-03'
   //

   //---------------------------------------------------------------
   // You can use %subst with the %char result if you only want
   // part of the result
   //---------------------------------------------------------------
   result = 'The time is now ' + %SUBST (%CHAR(time):1:5) + '.';
   // result = 'The time is now 12:23.'

   //---------------------------------------------------------------
   // Use %CHAR to convert a graphic value to character so it
   // can be concatenated with a character value.
   //---------------------------------------------------------------
   result = 'The customer''s name is ' + %CHAR(Name) + '.';
   // result = 'The customer's name is XXYYZZ.'

   //-----------------------------------------------------
   // Use %CHAR to convert a number to character format:
   //-----------------------------------------------------
   result = 'You have ' + %char(points) + ' points.';
   // result = 'You have 234 points.'
   //
 /END-FREE
```

Figure 141. %CHAR Examples

%CHECK (Check Characters)

```
%CHECK(comparator : base {: start})
```

%CHECK returns the first position of the string *base* that contains a character that does not appear in string *comparator*. If all of the characters in *base* also appear in *comparator*, the function returns 0.

The check begins at the starting position and continues to the right until a character that is not contained in the comparator string is found. The starting position defaults to 1.

The first parameter must be of type character, graphic, or UCS-2, fixed or varying length. The second parameter must be the same type as the first parameter. The third parameter, if specified, must be a non-float numeric with zero decimal positions.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *-------------------------------------------------
 * A string contains a series of numbers separated
 * by blanks and/or commas.
 * Use %CHECK to extract the numbers
 *-------------------------------------------------
D string          s             50a   varying
D                                     inz('12, 233 17, 1, 234')
D delimiters      C                   ' ,'
D digits          C                   '0123456789'
D num             S             50a   varying
D pos             S             10i 0
D len             S             10i 0
D token           s             50a   varying

 /free

     // make sure the string ends with a delimiter
     string = string + delimiters;

     dou string = '';

       // Find the beginning of the group of digits
       pos = %check (delimiters : string);
       if (pos = 0);
          leave;
       endif;

       // skip past the delimiters
       string = %subst(string : pos);

       // Find the length of the group of digits
       len = %check (digits : string) - 1;

       // Extract the group of digits
       token = %subst(string : 1 : len);
       dsply ' ' ' ' token;

       // Skip past the digits
       if (len < %len(string));
         string = %subst (string : len + 1);
       endif;

     enddo;

 /end-free
```

Figure 142. %CHECK Example

See also Figure 144 on page 421.

%CHECKR (Check Reverse)

```
%CHECKR(comparator : base {: start})
```

%CHECKR returns the last position of the string *base* that contains a character that does not appear in string *comparator*. If all of the characters in *base* also appear in *comparator*, the function returns 0.

The check begins at the starting position and continues to the left until a character that is not contained in the comparator string is found. The starting position defaults to the end of the string.

The first parameter must be of type character, graphic, or UCS-2, fixed or varying length. The second parameter must be the same type as the first parameter. The third parameter, if specified, must be a non-float numeric with zero decimal positions.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
*-------------------------------------------
* If a string is padded at the end with some
* character other than blanks, the characters
* cannot be removed using %TRIM.
* %CHECKR can be used for this by searching
* for the last character in the string that
* is not in the list of "pad characters".
*-------------------------------------------
D string1         s             50a   varying
D                                     inz('My *dog* Spot.* @ * @ *')
D string2         s             50a   varying
D                                     inz('someone@somewhere.com')
D padChars        C                   ' *@'

 /free

   %len(string1) = %checkr(padChars:string1);
   //  %len(string1) is set to 14 (the position of the last character
   //  that is not in "padChars").

   //  string1 = 'My *dog* Spot.'

   %len(string2) = %checkr(padChars:string2);
   //  %len(string2) is set to 21 (the position of the last character
   //  that is not in "padChars").

   //  string2 = 'someone@somewhere.com'  (the string is not changed)

 /end-free
```

Figure 143. %CHECKR Example

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *------------------------------------------------------
 * A string contains a numeric value, but it might
 * be surrounded by blanks and asterisks and might be
 * preceded by a currency symbol.
 *------------------------------------------------------
D string          s             50a   varying inz('$****12.345***  ')

 /free
    // Find the position of the first character that is not one of ' $*'
    numStart = %CHECK (' $*' : string);
    //  = 6

    // Find the position of the last character that is not one of ' *'
    numEnd = %CHECKR (' *' : string);
    //  = 11

    // Extract the numeric string
    string = %SUBST(string : numStart : numEnd - numStart + 1);
    //  = '12.345'

 /end-free
```

Figure 144. %CHECK and %CHECKR Example

%DATE (Convert to Date)

```
%DATE{(expression{:date-format})}
```

%DATE converts the value of the expression from character, numeric, or timestamp data to type date. The converted value remains unchanged, but is returned as a date.

The first parameter is the value to be converted. If you do not specify a value, %DATE returns the current system date.

The second parameter is the date format for character or numeric input. Regardless of the input format, the output is returned in *ISO format.

For information on the input formats that can be used, see “Date Data” on page 119. If the date format is not specified for character or numeric input, the default value is either the format specified on the DATFMT control-specification keyword or *ISO. For more information, see “DATFMT(fmt{separator})” on page 227.

If the first parameter is a timestamp, *DATE, or UDATE, do not specify the second parameter. The system knows the format of the input in these cases.

For more information, see “Information Operations” on page 366 or Chapter 25, “Built-In Functions,” on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

  string = '040596';
  date =  %date(string:*MDY0);
  // date now contains d'1996-04-05'
/END-FREE
```

Figure 145. %DATE Example

%DAYS (Number of Days)

```
%DAYS(number)
```

%DAYS converts a number into a duration that can be added to a date or timestamp value.

%DAYS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of days added or subtracted. For a date, the resulting value is in *ISO format.

For an example of date and time arithmetic operations, see Figure 173 on page 458.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

%DEC (Convert to Packed Decimal Format)

```
%DEC(numeric or character expression{:precision:decimal places})
%DEC(date time or timestamp expression {:format})
```

%DEC converts the value of the first parameter to decimal (packed) format.

Numeric or character expression

When the first parameter is a numeric or character expression, the result has *precision* digits and *decimal places* decimal positions. The precision and decimal places must be numeric literals, named constants that represent numeric literals, or built-in functions with a numeric value known at compile-time.

Note: %LEN and %DECPOS cannot be used directly for the second and third parameters of %DEC or %DECH, even if the values of %LEN and %DECPOS are constant. See Figure 170 on page 453 for an example using the length and decimal positions of a variable to control %DEC and %DECH.

Parameters *precision* and *decimal places* may be omitted if the type of expression is neither float nor character. If these parameters are omitted, the precision and decimal places are taken from the attributes of the numeric expression.

If the parameter is a character expression, the following rules apply:
- The sign is optional. It can be '+' or '-'. It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Blanks are allowed anywhere in the data. For example, ' + 3 ' is a valid parameter.
- The second and third parameters are required.
- Floating point data, for example '1.2E6', is not allowed.
- If invalid numeric data is found, an exception occurs with status code 105.

See %DECHfor examples using %DEC.

Date, time or timestamp expression

When the first parameter is a date time or timestamp expression, the optional format parameter specifies the format of the value returned. The converted decimal value will have the number of digits that a value of that format can have, and zero decimal positions. For example, if the first parameter is a date, and the format is *YMD, the decimal value will have six digits.

If the format parameter is omitted, the format of the first parameter is used. See "DATFMT(fmt{separator})" on page 227 and "TIMFMT(fmt{separator})" on page 235.

Format *USA is not allowed with a time expression. If the first parameter is a time value with a time-format of *USA, the second format parameter for %DEC must be specified.

Figure 147 on page 426 shows an example of the %DEC built-in function.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
D   yyddd         S              5S 0
D   yyyymmdd      S              8P 0
D   hhmmss        S              6P 0
D   numeric       S             20S 0
D   date          S               D   inz(D'2003-06-27') DATFMT(*USA)
D   time          S               T   inz(T'09.25.59')
D   timestamp     S               Z   inz(Z'2003-06-27-09.25.59.123456'
 /free

     // Using the format of the first parameter

     numeric = %dec(date);          //   numeric = 06272003
     numeric = %dec(time);           //   numeric = 092559
     numeric = %dec(timestamp);      //   numeric = 20030627092559123456

     // Using the second parameter to specify the result format

     yyddd = %dec(date : *jul);      //   yyddd = 03178
     yyyymmdd = %dec(date : *iso);   //   yyyymmdd = 20030627
```

Figure 146. Using %DEC to convert dates, times and timestamps to numeric

%DECH (Convert to Packed Decimal Format with Half Adjust)

```
%DECH(numeric or character expression :precision:decimal places )
```

%DECH is the same as %DEC except that if the expression is a decimal or float value, half adjust is applied to the value of the expression when converting to the desired precision. No message is issued if half adjust cannot be performed..

Unlike, %DEC, all three parameters are required.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

%DECH Examples

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D p7              s              7p 3 inz (1234.567)
D s9              s              9s 5 inz (73.73442)
D f8              s              8f   inz (123.456789)
D c15a            s             15a   inz (' 123.456789 -')
D c15b            s             15a   inz (' + 9 , 8 7 6 ')
D result1         s             15p 5
D result2         s             15p 5
D result3         s             15p 5

 /FREE

// using numeric parameters
    result1 = %dec (p7) + 0.011;  // "result1" is now 1234.57800
    result2 = %dec (s9 : 5: 0);   // "result2" is now   73.00000
    result3 = %dech (f8: 5: 2);   // "result3" is now  123.46000
// using character parameters
     result1 = %dec (c15a: 5: 2);  // "result1" is now -123.45
     result2 = %dech(c15b: 5: 2);  // "result2" is now    9.88000
 /END-FREE
```

Figure 147. Using Numeric and Character Parameters

```
*-----------------------------------------------------------------
* If the character data is known to contain non-numeric characters
* such as thousands separators (like 1,234,567) or leading
* asterisks and currency symbols (like $***1,234,567.89), some
* preprocessing is necessary to remove these characters from the
* data.
*-----------------------------------------------------------------


D data            s             20a   inz('$1,234,567.89')
D num             s             21p 9
 /free
    // Use the %XLATE builtin function to replace any currency
    // symbol, asterisks or thousands separators with blanks
    num = %dech(%xlate('$*,' : '   ' : data)
              : 21 : 9);
    // If the currency symbol or thousands separator might
    // vary at runtime, use variables to hold these values.
    num = %dech(%xlate(cursym + '*' + thousandsSep : '   ' : data)
              : 21 : 9);
```

Figure 148. Handling Currency Symbols and Thousands Separators

%DECPOS (Get Number of Decimal Positions)

```
%DECPOS(numeric expression)
```

%DECPOS returns the number of decimal positions of the numeric variable or expression. The value returned is a constant, and so may participate in constant folding.

The numeric expression must not be a float variable or expression.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D p7              s              7p 3 inz (8236.567)
D s9              s              9s 5 inz (23.73442)
D result1         s              5i 0
D result2         s              5i 0
D result3         s              5i 0

 /FREE
   result1 = %decpos (p7);     //  "result1" is now  3.
   result2 = %decpos (s9);     //  "result2" is now  5.
   result3 = %decpos (p7 * s9);//  "result3" is now  8.
 /END-FREE
```

Figure 149. %DECPOS Example

See Figure 170 on page 453 for an example of %DECPOS with %LEN.

For more information, see “Size Operations” on page 375 or Chapter 25, “Built-In Functions,” on page 399.

%DIFF (Difference Between Two Date, Time, or Timestamp Values)

```
%DIFF(op1:op2:*MSECONDS|*SECONDS|*MINUTES|*HOURS|*DAYS|*MONTHS|*YEARS)
%DIFF(op1:op2:*MS|*S|*MN|*H|*D|*M|*Y)
```

%DIFF produces the difference (duration) between two date or time values. The first and second parameters must have the same, or compatible types. The following combinations are possible:

- Date and date
- Time and time
- Timestamp and timestamp
- Date and timestamp (only the date portion of the timestamp is considered)
- Time and timestamp (only the time portion of the timestamp is considered).

The third parameter specifies the unit. The following units are valid:

- For two dates or a date and a timestamp: *DAYS, *MONTHS, and *YEARS
- For two times or a time and a timestamp: *SECONDS, *MINUTES, and *HOURS
- For two timestamps: *MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, and *YEARS

The result is rounded down, with any remainder discarded. For example, 61 minutes is equal to 1 hour, and 59 minutes is equal to 0 hours.

The value returned by the function is compatible with both type numeric and type duration. You can add the result to a number (type numeric) or a date, time, or timestamp (type duration).

If you ask for the difference in microseconds between two timestamps that are more than 32 years 9 months apart, you will exceed the 15-digit limit for duration values. This will result in an error or truncation.

For more information, see “Date Operations” on page 359 or Chapter 25, “Built-In Functions,” on page 399.

```
D due_date        S               D   INZ(D'2005-06-01')
D today           S               D   INZ(D'2004-09-23')
D num_days        S            15P 0

D start_time      S               Z
D time_taken      S            15P 0

 /FREE

     // Determine the number of days between two dates.

     // If due_date has the value 2005-06-01 and
     // today has the value 2004-09-23, then
     // num_days will have the value 251.

     num_days = %DIFF (due_date: today: *DAYS);

     // If the arguments are coded in the reverse order,
     // num_days will have the value -251.

     num_days = %DIFF (today: due_date: *DAYS);

     // Determine the number of seconds required to do a task:
     //  1. Get the starting timestamp
     //  2. Do the task
     //  3. Calculate the difference between the current
     //     timestamp and the starting timestamp

     start_time = %timestamp();
     process();
     time_taken = %DIFF (%timestamp() : start_time : *SECONDS);

 /END-FREE
```

Figure 150. Using the result of %DIFF as a numeric value

```
D estimated_end...
D                 S              D
D prev_start      S              D   INZ(D'2003-06-21')
D prev_end        S              D   INZ(D'2003-06-24')

 /FREE

     // Add the number of days between two dates
     // to a third date

     // prev_start is the date a previous task began
     // prev_end is the date a previous task ended.

     // The following calculation will estimate the
     // date a similar task will end, if it begins
     // today.

     // If the current date, returned by %date(), is
     // 2003-08-15, then estimated_end will be
     // 2003-08-18.

     estimated_end = %date() + %DIFF(prev_end : prev_start : *days);

 /END-FREE
```

Figure 151. Using the result of %DIFF as a duration

%DIV (Return Integer Portion of Quotient)

```
%DIV(n:m)
```

%DIV returns the integer portion of the quotient that results from dividing operands **n** by **m**. The two operands must be numeric values with zero decimal positions. If either operand is a packed, zoned, or binary numeric value, the result is packed numeric. If either operand is an integer numeric value, the result is integer. Otherwise, the result is unsigned numeric. Float numeric operands are not allowed. (See also "%REM (Return Integer Remainder)" on page 465.)

If the operands are constants that can fit in 8-byte integer or unsigned fields, constant folding is applied to the built-in function. In this case, the %DIV built-in function can be coded in the definition specifications.

For more information, see "Arithmetic Operations" on page 348 or Chapter 25, "Built-In Functions," on page 399.

This function is illustrated in Figure 179 on page 465.

%EDITC (Edit Value Using an Editcode)

```
%EDITC(numeric : editcode {: *ASTFILL | *CURSYM | currency-symbol})
```

This function returns a character result representing the numeric value edited according to the edit code. In general, the rules for the numeric value and edit code are identical to those for editing numeric values in output specifications. The third parameter is optional, and if specified, must be one of:

***ASTFILL**
Indicates that asterisk protection is to be used. This means that leading zeroes are replaced with asterisks in the returned value. For example, %EDITC(-0012.5 : 'K' : *ASTFILL) returns '**12.5–'.

***CURSYM**
Indicates that a floating currency symbol is to be used. The actual symbol will be the one specified on the control specification in the CURSYM keyword, the the default '$'. When *CURSYM is specified, the currency symbol is placed in the result just before the first significant digit. For example, %EDITC(0012.5 : 'K' : *CURSYM) returns ' $12.5 '.

currency–symbol
Indicates that floating currency is to be used with the provided currency symbol. It must be a 1–byte character constant (literal, named constant, or expression that can be evaluated at compile time).For example, %EDITC(0012.5 : 'K' : 'X') returns ' X12.5 '.

The result of %EDITC is always the same length, and may contain leading and trailing blanks. For example, %EDITC(NUM : 'A' : '$') might return '$1,234.56CR' for one value of NUM and ' $4.56 ' for another value.

Float expressions are not allowed in the first parameter (you can use %DEC to convert a float to an editable format). The edit code is specified as a character constant; supported edit codes are: 'A' – 'D', 'J' – 'Q', 'X' – 'Z' , '1' – '9'. The constant can be a literal, named constant or an expression whose value can be determined at compile time.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D msg             S            100A
D salary          S              9P 2 INZ(1000)
 * If the value of salary is 1000, then the value of salary * 12
 * is 12000.00. The edited version of salary * 12 using the A edit
 * code with floating currency is ' $12,000.00 '.
 * The value of msg is 'The annual salary is $12,000.00'
CL0N01Factor1+++++++Opcode&ExtExtended-factor2++++++++++++++++++++++++++++
C                   EVAL      msg = 'The annual salary is '
C                                 + %trim(%editc(salary * 12
C                                                :'A': *CURSYM))
 * In the next example, the value of msg is 'The annual salary is &12,000.00'
C                   EVAL      msg = 'The annual salary is '
C                                 + %trim(%editc(salary * 12
C                                                :'A': '&'))

 * In the next example, the value of msg is 'Salary is $*****12,000.00'
 * Note that the '$' comes from the text, not from the edit code.
C                   EVAL      msg = 'Salary is $'
C                                 + %trim(%editc(salary * 12
C                                                 :'B': *ASTFILL))

 * In the next example, the value of msg is 'The date is 1/14/1999'
C                   EVAL      msg = 'The date is '
C                                 + %trim(%editc(*date : 'Y'))
```

Figure 152. %EDITC Example 1

A common requirement is to edit a field as follows:

- Leading zeros are suppressed
- Parentheses are placed around the value if it is negative

The following accomplishes this using an %EDITC in a subprocedure:

```
D neg             S              5P 2      inz(-12.3)
D pos             S              5P 2      inz(54.32)
D editparens      PR            50A
D    val                        30P 2      value
D editedVal       S             10A

C                   EVAL      editedVal = editparens(neg)

C* Now editedVal has the value '(12.30)   '

C                   EVAL      editedVal = editparens(pos)

C* Now editedVal has the value ' 54.32    '

 *---------------------------------------------------------------
 * Subprocedure EDITPARENS
 *---------------------------------------------------------------

P editparens      B
D editparens      PI            50A
D    val                        30P 2      value
D lparen          S              1A        inz(' ')
D rparen          S              1A        inz(' ')
D res             S             50A

C* Use parentheses if the value is negative

C                   IF        val < 0
C                   EVAL      lparen = '('
C                   EVAL      rparen = ')'
C                   ENDIF

C* Return the edited value
C* Note that the '1' edit code does not include a sign so we
C* don't have to calculate the absolute value.

C                   RETURN    lparen             +
C                             %editc(val : '1')  +
C                             rparen

P editparens      E
```

Figure 153. %EDITC Example 2

%EDITFLT (Convert to Float External Representation)

```
%EDITFLT(numeric expression)
```

%EDITFLT converts the value of the numeric expression to the character external display representation of float. The result is either 14 or 23 characters. If the argument is a 4-byte float field, the result is 14 characters. Otherwise, it is 23 characters.

If specified as a parameter to a definition specification keyword, the parameter must be a numeric literal, float literal, or numeric valued constant name or built-in function. When specified in an expression, constant folding is applied if the numeric expression has a constant value.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D f8              s              8f   inz (50000)
D string          s             40a   varying

 /FREE
    string = 'Float value is ' + %editflt (f8 - 4E4) + '.';
    // Value of "string" is 'Float value is +1.000000000000000E+004. '
 /END-FREE
```

Figure 154. %EDITFLT Example

%EDITW (Edit Value Using an Editword)

```
%EDITW(numeric : editword)
```

This function returns a character result representing the numeric value edited according to the edit word. The rules for the numeric value and edit word are identical to those for editing numeric values in output specifications.

Float expressions are not allowed in the first parameter. Use %DEC to convert a float to an editable format.

The edit word must be a character constant.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D amount          S             30A
D salary          S              9P 2
D editwd          C                   '$ ,   ,  **Dollars&  &Cents'

 * If the value of salary is 2451.53, then the edited version of
 * (salary * 12) is '$***29,418*Dollars 36 Cents'. The value of
 * amount is 'The annual salary is $***29,418*Dollars 36 Cents'.

 /FREE
   amount = 'The annual salary is '
                + %editw(salary * 12 : editwd);
 /END-FREE
```

Figure 155. %EDITW Example

%ELEM (Get Number of Elements)

```
%ELEM(table_name)
%ELEM(array_name)
%ELEM(multiple_occurrence_data_structure_name)
```

%ELEM returns the number of elements in the specified array, table, or multiple-occurrence data structure. It may be specified anywhere a numeric constant is allowed in the definition specification or in an expression in the extended factor 2 field.

The parameter must be the name of an array, table, or multiple occurrence data structure.

For more information, see "Array Operations" on page 351 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D arr1d           S             20    DIM(10)
D table           S             10    DIM(20) ctdata
D mds             DS            20    occurs(30)
D num             S              5p 0

 * like_array will be defined with a dimension of 10.
 * array_dims will be defined with a value of 10.
D like_array      S                   like(arr1d) dim(%elem(arr1d))
D array_dims      C                   const (%elem (arr1d))

 /FREE
    num = %elem (arr1d);  // num is now 10
    num = %elem (table);  // num is now 20
    num = %elem (mds);    // num is now 30
 /END-FREE
```

Figure 156. %ELEM Example

%EOF (Return End or Beginning of File Condition)

```
%EOF{(file_name)}
```

%EOF returns '1' if the most recent read operation or write to a subfile ended in an end of file or beginning of file condition; otherwise, it returns '0'.

The operations that set %EOF are:
- "READ (Read a Record)" on page 653
- "READC (Read Next Changed Record)" on page 656
- "READE (Read Equal Key)" on page 658
- "READP (Read Prior Record)" on page 661
- "READPE (Read Prior Equal)" on page 663
- "WRITE (Create New Records)" on page 717 (subfile only).

The following operations, if successful, set %EOF(filename) off. If the operation is not successful, %EOF(filename) is not changed. %EOF with no parameter is not changed by these operations.
- "CHAIN (Random Retrieval from a File)" on page 529
- "OPEN (Open File for Processing)" on page 642
- "SETGT (Set Greater Than)" on page 679
- "SETLL (Set Lower Limit)" on page 681

When a full-procedural file is specified, this function returns '1' if the previous operation in the list above, for the specified file, resulted in an end of file or beginning of file condition. For primary and secondary files, %EOF is available only if the file name is specified. It is set to '1' if the most recent input operation during *GETIN processing resulted in an end of file or beginning of file condition. Otherwise, it returns '0'.

This function is allowed for input, update, and record-address files; and for display files allowing WRITE to subfile records.

For more information, see "File Operations" on page 363 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * File INFILE has record format INREC
FINFILE    IF   E             DISK         remote

 /FREE
    READ INREC;  // read a record
    IF  %EOF;
                 // handle end of file
    ENDIF;
 /END-FREE
```

Figure 157. %EOF without a Filename Parameter

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 * This program is comparing two files

F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
FFILE1     IF   E             DISK       remote
FFILE2     IF   E             DISK       remote

 * Loop until either FILE1 or FILE2 has reached end-of-file
 /FREE
    DOU %EOF(FILE1) OR %EOF(FILE2);
       // Read a record from each file and compare the records

       READ REC1;
       READ REC2;
       IF %EOF(FILE1) AND %EOF(FILE2);
          // Both files have reached end-of-file
          EXSR EndCompare;

       ELSEIF %EOF(FILE1);
          // FILE1 is shorter than FILE2
          EXSR F1Short;

       ELSEIF %EOF(FILE2);
          // FILE2 is shorter than FILE1
          EXSR F2Short;

       ELSE;
          // Both files still have records to be compared
          EXSR CompareRecs;
       ENDIF;
    ENDDO;
  // ...
 /END-FREE
```

Figure 158. %EOF with a Filename Parameter

%EQUAL (Return Exact Match Condition)

```
%EQUAL{(file_name)}
```

%EQUAL returns '1' if the most recent relevant operation found an exact match; otherwise, it returns '0'.

The operations that set %EQUAL are:
- "SETLL (Set Lower Limit)" on page 681
- "LOOKUP (Look Up a Table or Array Element)" on page 599

If %EQUAL is used without the optional file_name parameter, then it returns the value set for the most recent relevant operation.

For the SETLL operation, this function returns '1' if a record is present whose key or relative record number is equal to the search argument.

For the LOOKUP operation with the EQ indicator specified, this function returns '1' if an element is found that exactly matches the search argument.

If a file name is specified, this function applies to the most recent SETLL operation for the specified file. This function is allowed only for files that allow the SETLL operation code.

For more information, see "File Operations" on page 363, "Result Operations" on page 375, or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * File CUSTS has record format CUSTREC
FCUSTSIF   E           K DISK       remote

 /FREE
    // Check if the file contains a record with a key matching Cust
    setll Cust CustRec;
    if %equal;
    // an exact match was found in the file
    endif;
 /END-FREE
```

Figure 159. %EQUAL with SETLL Example

```
     DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
     D TabNames        S             10A   DIM(5) CTDATA ASCEND
     D SearchName      S             10A
      * Position the table at or near SearchName
      * Here are the results of this program for different values
      * of SearchName:
      *   SearchName   |   DSPLY
      *   -------------+------------------------------
      *   'Catherine ' |   'Next greater     Martha'
      *   'Andrea    ' |   'Exact            Andrea'
      *   'Thomas    ' |   'Not found        Thomas'
     C..N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
     C     SearchName    LOOKUP    TabNames                            10  10
     C                   SELECT
     C                   WHEN      %EQUAL
      * An exact match was found
     C                   WHEN      %FOUND
      * A name was found greater than SearchName
     C                   OTHER
      * Not found.  SearchName is greater than all the names in the table
     C                   ENDSL
     C                   RETURN
**CTDATA TabNames
Alexander
Andrea
Bohdan
Martha
Samuel
```

Figure 160. %EQUAL and %FOUND with LOOKUP Example

%ERROR (Return Error Condition)

%ERROR returns '1' if the most recent operation with extender 'E' specified resulted in an error condition. This is the same as the error indicator being set on for the operation. Before an operation with extender 'E' specified begins, %ERROR is set to return '0' and remains unchanged following the operation if no error occurs. All operations that allow an error indicator can also set the %ERROR built-in function.

For examples of the %ERROR built-in function, see Figure 185 on page 476 and Figure 186 on page 477.

For more information, see "Result Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

%FIELDS (Fields to update)

```
%FIELDS(name{:name...})
```

A list of fields can be specified as the final argument to Input/Output operation UPDATE coded in a free-form group. Only the fields specified are updated into the Input/Output buffer.

Notes:

1. Each name must be the name of a field in the input buffer for the record. If the field is renamed, the internal name is used.

%FIELDS specifies a list of fields to update. For example:

Figure 161. Updating Fields

```
/free
  chain empno record;
  salary = salary + 2000;
  status = STATEXEMPT;
  update record %fields(salary:status);
/end-free
```

%FLOAT (Convert to Floating Format)

```
%FLOAT(numeric or character expression)
```

%FLOAT converts the value of the expression to float format. This built-in function may only be used in expressions.

If the parameter is a character expression, the following rules apply:
- The sign is optional. It can be '+' or '-'. It must precede the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- The exponent is optional. It can be either 'E' or 'e'. The sign for the exponent is optional. It must precede the numeric part of the exponent.
- Blanks are allowed anywhere in the data. For example, ' + 3 , 5 E 9' is a valid parameter.
- If invalid numeric data is found, an exception occurs with status code 105.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D p1              s             15p 0 inz (1)
D p2              s             25p13 inz (3)
D c15a            s             15a   inz('-5.2e-1')
D c15b            s             15a   inz(' + 5 . 2 ')
D result1         s             15p 5
D result2         s             15p 5
D result3         s             15p 5
D result4         s              8f
 /FREE
 // using numeric parameters
    result1 = p1 / p2;          //  "result1" is now  0.33000.
    result2 = %float (p1) / p2; //  "result2" is now  0.33333.
    result3 = %float (p1 / p2); //  "result3" is now  0.33333.
   result4 = %float (12345);   //  "result4" is now 1.2345E4
 // using character parameters
   result1 = %float (c15a);    //  "result1" is now -0.52000.
   result2 = %float (c15b);    //  "result2" is now  5.20000.
   result4 = %float (c15b);    //  "result4" is now  5.2E0
 /END-FREE
```

Figure 162. %FLOAT Example

%FOUND (Return Found Condition)

```
%FOUND{(file_name)}
```

%FOUND returns '1' if the most recent relevant file operation found a record, a string operation found a match, or a search operation found an element. Otherwise, this function returns '0'.

If %FOUND is used without the optional file_name parameter, then it returns the value set for the most recent relevant operation. When a file_name is specified, then it applies to the most recent relevant operation on that file.

For file operations, %FOUND is opposite in function to the "no record found NR" indicator.

For string operations, %FOUND is the same in function as the "found FD" indicator.

For the LOOKUP operation, %FOUND returns '1' if the operation found an element satisfying the search conditions.

For more information, see "File Operations" on page 363, "Result Operations" on page 375, or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * File CUSTS has record format CUSTREC
FCUSTS     IF   E           K DISK       remote

 /FREE
    // Check if the customer is in the file
    chain Cust CustRec;
    if %found;
       exsr HandleCustomer;
    endif;
 /END-FREE
```

Figure 163. %FOUND used to Test a File Operation without a Parameter

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * File MASTER has all the customers
 * File GOLD has only the "privileged" customers
FMASTER    IF   E           K DISK         remote
FGOLD      IF   E           K DISK         remote

 /FREE
   // Check if the customer exists, but is not a privileged customer
   chain Cust MastRec;
   chain Cust GoldRec;

   // Note that the file name is used for %FOUND, not the record name
   if  %found (Master)  and not  %found (Gold);
   //
   endif;
 /END-FREE
```

Figure 164. %FOUND used to Test a File Operation with a Parameter

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D Numbers         C                   '0123456789'
D Position        S              5I 0
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * If the actual position of the name is not required, just use
 * %FOUND to test the results of the SCAN operation.
 * If Name has the value 'Barbara' and Line has the value
 * 'in the city of Toronto.      ', then %FOUND will return '0'.
 * If Line has the value 'the city of Toronto where Barbara lives, '
 * then %FOUND will return '1'.
C     Name          SCAN      Line
C                   IF        %FOUND
C                   EXSR      PutLine
C                   ENDIF
 * If Value contains the value '12345.67', Position would be set
 * to 6 and %FOUND would return the value '1'.
 * If Value contains the value '10203040', Position would be set
 * to 0 and %FOUND would return the value '0'.
C     Numbers       CHECK     Value         Position
C                   IF        %FOUND
C                   EXSR      HandleNonNum
C                   ENDIF
```

Figure 165. %FOUND used to Test a String Operation

%GETATR (Retrieve Attribute)

```
%GETATR(window_name:part_name:attribute_name)
```

%GETATR returns the attribute value of a part on a window. Both the first and second parameters can be %WINDOW or %PART.

For an alternative form of accessing part attributes, see "Qualified GUI Part Attribute Access" on page 379.

Notes:

1. The %GETATR built-in function does not affect the corresponding program fields for parts. If you want the corresponding program field for the part to contain the current value of an entry field, make it the target of the %GETATR built-in, for example:

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments++++++
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++++Comments++++++
C                   EVAL      ENT0000B = %GETATR('INVENTORY':'ENT0000B':'TEXT')
```

Figure 166. %GETATR Example

2. The %GETATR built-in function does not support 1-byte and 8-byte signed and unsigned integer values, and unicode values.

%GRAPH (Convert to Graphic Value)

```
%GRAPH(char-expr | graph-expr | UCS-2-expr { : ccsid })
```

%GRAPH converts the value of the expression from character, graphic, or UCS-2 and returns a graphic value. The result is varying length if the parameter is varying length.

The second parameter, *ccsid*, is optional and indicates the CCSID of the resulting expression. The CCSID defaults to the graphic CCSID related to the workstation CCSID. If CCSID(*GRAPH : *IGNORE) is specified on the control specification or assumed for the module, the %GRAPH built-in is not allowed.

If the parameter is a constant, the conversion will be done at compile time. In this case, the CCSID is the graphic CCSID related to the CCSID of the source file.

If the conversion results in substitution characters, a warning message is issued at compile time. At run time, status 00050 is set and no error message is issued.

Note: Conversions between 2 unicode CCSIDs are not supported. For a list of supported CCSID values see Appendix C, "Supported CCSID Values," on page 735

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
H*Keywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H ccsid (*graph: 942)

D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++
D  char           S              6A   inz('XXYYZZ')
 * The %GRAPH built-in function is used to initialize a graphic field
D  graph          S             10G   inz (%graph ('AABBCCDDEE'))
D  ufield         S              2C   inz (%ucs2 ('FFGG'))
D  graph2         S              2G   ccsid (951) inz (*hival)
D  isEqual        S              1N
D  proc           PR
D    gparm                       2G   ccsid (951) value

 /FREE
    graph = %graph (char) + %graph (ufield);
    // graph now has the value XXYYZZFFGG.

    isEqual = graph = %graph (graph2 : 942);
    // The result of the %GRAPH built-in function is the value of
    // graph2, converted from CCSID  951 to CCSID 942.

    graph2 = graph;
    // The value of graph is converted from CCSID 942 to CCSID 951
    // and stored in graph2.
    // This conversion is performed implicitly by the compiler.

    proc (graph);
    // The value of graph is converted from CCSID 942 to CCSID 951
    // implicitly, as part of passing the parameter by value.
 /END-FREE
```

Figure 167. %GRAPH Examples

%HOURS (Number of Hours)

```
%HOURS(number)
```

%HOURS converts a number into a duration that can be added to a time or timestamp value.

%HOURS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of hours added or subtracted. For a time, the resulting value is in *ISO format.

For an example of date and time arithmetic operations, see Figure 173 on page 458.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

%INT (Convert to Integer Format)

```
%INT(numeric or character expression)
```

%INT converts the value of the expression to integer. Any decimal digits are truncated. This built-in function may only be used in expressions. %INT can be used to truncate the decimal positions from a float or decimal value allowing it to be used as an array index.

If the parameter is a character expression, the following rules apply:
- The sign is optional. It can be '+' or '-'. It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Blanks are allowed anywhere in the data. For example, ' + 3 ' is a valid parameter.
- Floating point data is not allowed. That is, where the numeric value is followed by E and an exponent, for example '1.2E6'.
- If invalid numeric data is found, an exception occurs with status code 105

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

Figure 168 on page 450 shows an example of the %INT built-in function.

%INTH (Convert to Integer Format with Half Adjust)

```
%INTH(numeric or character expression)
```

%INTH is the same as %INT except that if the expression is a decimal, float or character value, half adjust is applied to the value of the expression when converting to integer type. No message is issued if half adjust cannot be performed.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

%INTH (Convert to Integer Format with Half Adjust)

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D p7              s              7p 3 inz (1234.567)
D s9              s              9s 5 inz (73.73442)
D f8              s              8f   inz (123.789)
D c15a            s             15a   inz (' 12345.6789 -')
D c15b            s             15a   inz (' + 9 8 7 . 6 5 4 ')
D result1         s             15p 5
D result2         s             15p 5
D result3         s             15p 5
D array           s              1a   dim (200)
D a               s              1a

 /FREE
 // using numeric parameters
    result1 = %int (p7) + 0.011;  // "result1" is now 1234.01100.
    result2 = %int (s9);          // "result2" is now   73.00000
    result3 = %inth (f8);         // "result3" is now  124.00000.
 // using character parameters
    result1 = %int (c15a);        // "result1" is now  -12345.00000
    result2 = %inth (c15b);       // "result2" is now     988.00000

    // %INT and %INTH can be used as array indexes
    a = array (%inth (f8));
 /END-FREE
```

Figure 168. %INT and %INTH Example

%KDS (Search Arguments in Data Structure)

```
%KDS(data-structure-name{:num-keys})
```

%KDS is allowed as the search argument for any keyed Input/Output operation (CHAIN, DELETE, READE, READPE, SETGT, SETLL) coded in a free-form group. The search argument is specified by the subfields of the data structure name coded as the first argument of the built-in function. The key data structure may be (but is not limited to), an externally described data structure with keyword EXTNAME(...:*KEY) or LIKEREC(...:*KEY)..

Notes:

1. The first argument must be the name of a data structure. This includes any subfield defined with keyword LIKEDS or LIKEREC.
2. The second argument specifies how many of the subfields to use as the search argument.
3. The individual key values in the compound key are taken from the top level subfields of the data structure. Subfields defined with LIKEDS are considered character data.
4. Subfields used to form the compound key must not be arrays.
5. The types of all subfields (up to the number specified by ″num-keys″) must match the types of the actual keys. Where lengths and formats differ, the value is converted to the proper length and format.
6. If the data structure is defined as an array data structure (using keyword DIM), an index must be supplied for the data structure.
7. Opcode extenders H, M, or R specified on the keyed Input/Output operations code affect the moving of the search argument to the corresponding position in the key build area.

Example:

```
A..........T.Name++++++RLen++TDpB......Functions++++++++++++++++++
A          R CUSTR
A            NAME         100A
A            ZIP           10A
A            ADDR         100A
A          K NAME
A          K ZIP
FFilename++IPEASF.....L.....A.Device+.Keywords++++++++++++++++++++++++++
Fcustfile  if   e           k disk    rename(CUSTR:custRec)
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D custRecKeys     ds                  likerec(custRec : *key)
 ...
 /free
        // custRecKeys is a qualified data structure
        custRecKeys.name = customer;
        custRecKeys.zip = zipcode;
        // the *KEY data structure is used as the search argument for CHAIN
        chain %kds(custRecKeys) custRec;
 /end-free
```

Figure 169. Example of Search on Keyed Input/Output Operations

%LEN (Get or Set Length)

```
%LEN(expression)
```

%LEN can be used to get the length of a variable expression or to set the current length of a variable-length field.

The parameter must not be a figurative constant.

For more information, see "Size Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

%LEN Used for its Value

When used on the right-hand side of an expression, this function returns the number of digits or characters of the variable expression.

For numeric expressions, the value returned represents the precision of the expression and not necessarily the actual number of significant digits. For a float variable or expression, the value returned is either 4 or 8. When the parameter is a numeric literal, the length returned is the number of digits of the literal.

For character, graphic, or UCS-2 expressions the value returned is the number of characters in the value of the expression. For variable-length values, such as the value returned from a built-in function or a variable-length field, the value returned by %LEN is the current length of the character, graphic, or UCS-2 value.

Note that if the parameter is a built-in function or expression that has a value computable at compile-time, the length returned is the actual number of digits of the constant value rather than the maximum possible value that could be returned by the expression.

For all other data types, the value returned is the number of bytes of the value.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D num1            S              7P 2
D NUM1_LEN        C                   %len(num1)
D NUM1_DECPOS     C                   %decpos(num1)
D num2            S               5S 1
D num3            S               5I 0 inz(2)
D chr1            S              10A   inz('Toronto   ')
D chr2            S              10A   inz('Munich    ')
D ptr             S                *

 * Numeric expressions:
 /FREE
   num1 = %len(num1);                 //  7
   num1 = %decpos(num2);              //  1
   num1 = %len(num1*num2);            // 12
   num1 = %decpos(num1*num2);         //  3
   // Character expressions:
   num1 = %len(chr1);                 // 10
   num1 = %len(chr1+chr2);            // 20
   num1 = %len(%trim(chr1));          //  7
   num1 = %len(%subst(chr1:1:num3) + ' ' + %trim(chr2));//  9
   // %len and %decpos can be useful with other built-in functions:
   // Although this division is performed in float, the result is
   // converted to the same precision as the result of the eval:
   // Note: %LEN and %DECPOS cannot be used directly with %DEC
   //       and %DECH, but they can be used as named constants
   num1 = 27 + %dec (%float(num1)/num3 : NUM1_LEN : NUM1_DECPOS);
   // Allocate sufficient space to hold the result of the catenation
   // (plus an extra byte for a trailing null character):
   num3 = %len (chr1 + chr2) + 1;
   ptr = %alloc (num3);
   %str (ptr: num3) = chr1 + chr2;
 /END-FREE
```

Figure 170. %DECPOS and %LEN Example

%LEN Used to Set the Length of Variable-Length Fields

When used on the left-hand side of an expression, this function sets the current length of a variable-length field. If the set length is greater than the current length, the characters in the field between the old length and the new length are set to blanks.

Note: %LEN can only be used on the left-hand-side of an expression when the parameter is variable length.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
 *
D city            S             40A   varying inz('North York')
D n1              S              5i 0

 * %LEN used to get the current length of a variable-length field:
 /FREE
    n1 = %len(city);
    // Current length, n1 = 10

    // %LEN used to set the current length of a variable-length field:
    %len (city) = 5;
    // city = 'North' (length is 5)

    %len (city) = 15;
    // city = 'North          '  (length is 15)
 /END-FREE
```

Figure 171. %LEN with Variable-Length Field Example

%LOOKUPxx (Look Up an Array Element)

```
%LOOKUP(arg : array {: startindex {: numelems}})
%LOOKUPLT(arg : array {: startindex {: numelems}})
%LOOKUPGE(arg : array {: startindex {: numelems}})
%LOOKUPGT(arg : array {: startindex {: numelems}})
%LOOKUPLE(arg : array {: startindex {: numelems}})
```

The following functions return the array index of the item in *array* that matches *arg* as follows:

%LOOKUP An exact match.

%LOOKUPLT The value that is closest to *arg* but less than *arg*.

%LOOKUPLE An exact match, or the value that is closest to *arg* but less than *arg*.

%LOOKUPGT
The value that is closest to *arg* but greater than *arg*.

%LOOKUPGE
An exact match, or the value that is closest to *arg* but greater than *arg*.

If no value matches the specified condition, zero is returned.

The search starts at index *startindex* and continues for *numelems* elements. By default, the entire array is searched.

The first two parameters can have any type but must have the same type. They do not need to have the same length or number of decimal positions. The third and fourth parameters must be non-float numeric values with zero decimal positions.

For %LOOKUPLT, %LOOKUPLE, %LOOKUPGT, and %LOOKUPGE, the array must be defined with keyword ASCEND or DESCEND.

Built-in functions %FOUND and %EQUAL are not set following a %LOOKUP operation.

| The %LOOKUPxx builtin functions use a binary search for sequenced arrays
| (arrays that have the ASCEND or DESCEND keyword specified).

Note: Unlike the LOOKUP operation code, %LOOKUP applies only to arrays. To look up a value in a table, use the %TLOOKUP built-in function.

For more information, see “Array Operations” on page 351 or Chapter 25, “Built-In Functions,” on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
  arr(1) = 'Cornwall';
  arr(2) = 'Kingston';
  arr(3) = 'London';
  arr(4) = 'Paris';
  arr(5) = 'Scarborough';
  arr(6) = 'York';

  n = %LOOKUP('Paris':arr);
  // n = 4

  n = %LOOKUP('Thunder Bay':arr);
  // n = 0 (not found)

  n = %LOOKUP('Kingston':arr:3);
  // n = 0 (not found after start index)

  n = %LOOKUPLE('Paris':arr);
  // n = 4

  n = %LOOKUPLE('Milton':arr);
  // n = 3

  n = %LOOKGT('Sudbury':arr);
  // n = 6

  n = %LOOKGT('Yorks':arr:2:4);
  // n = 0 (not found between elements 2 and 5)
/END-FREE
```

Figure 172. %LOOKUPxx Example

Sequenced arrays that are not in the correct sequence

When the data is not in the correct sequence for a sequenced array, the %LOOKUPxx builtin functions and the LOOKUP operation code may find different values. The %LOOKUPxx builtin functions may not find a data value even if it is present in the array.

Since a binary search is used by the %LOOKUPxx builtin functions for a sequenced array, and the correct function of a binary search depends on the data being in order, the search may only look at a few elements of the array. When the array is out of order, the result of a binary search is unpredictable.

Note: When the LOOKUP operation code is used to find an exact match in a sequenced array, the search starts from the specified element and continues one element at a time until either the value is found or the last element of the array is reached.

%MINUTES (Number of Minutes)

```
%MINUTES(number)
```

%MINUTES converts a number into a duration that can be added to a time or timestamp value.

%MINUTES can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of minutes added or subtracted. For a time, the resulting value is in *ISO format.

For an example of date and time arithmetic operations, see Figure 173 on page 458.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

%MONTHS (Number of Months)

```
%MONTHS(number)
```

%MONTHS converts a number into a duration that can be added to a date or timestamp value.

%MONTHS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of months added or subtracted. For a date, the resulting value is in *ISO format.

In most cases, the result of adding or subtracting a given number of months is obvious. For example, 2000-03-15 + %MONTHS(1) is 2000-04-15. If the addition or subtraction would produce a nonexistent date (for example, February 30), the last day of the month is used instead.

Adding or subtracting a number of months to the 29th, 30th, or 31st day of a month may not be reversible. For example, 2000-03-31 + %MONTHS(1) - %MONTHS(1) is 2000-03-30.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

    // Determine the date in 3 years
    newdate = date + %YEARS(3);

    // Determine the date in 6 months prior
    loandate = duedate - %MONTHS(6);

    // Construct a timestamp from a date and time
    duestamp = duedate + t'12.00.00';
/END-FREE
```

Figure 173. %MONTHS and %YEARS Example

%MSECONDS (Number of Microseconds)

```
%MSECONDS(number)
```

%MSECONDS converts a number into a duration that can be added to a time or timestamp value.

%MSECONDS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of microseconds added or subtracted. For a time, the resulting value is in *ISO format.

For an example of date and time arithmetic operations, see Figure 173 on page 458.

For more information, see “Date Operations” on page 359 or Chapter 25, “Built-In Functions,” on page 399.

%NULLIND (Query or Set Null Indicator)

```
%NULLIND(fieldname)
```

The %NULLIND built-in function can be used to query or set the null indicator for null-capable fields. This built-in function can only be used if the User control compile option or ALWNULL(*USRCTL) keyword is specified. The fieldname can be a null-capable array element, data structure, stand-alone field, subfield, or multiple occurrence data structure.

%NULLIND can only be used in expressions in extended factor 2.

When used on the right-hand side of an expression, this function returns the setting of the null indicator for the null-capable field. The setting can be *ON or *OFF.

When used on the left-hand side of an expression, this function can be used to set the null indicator for null-capable fields to *ON or *OFF. The content of a null-capable field remains unchanged.

See “Database Null Value Support” on page 137 for more information on handling records with null-capable fields and keys.

For more information, see “Indicator-Setting Operations” on page 366 or Chapter 25, “Built-In Functions,” on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* Test the null indicator for a null-capable field.
/FREE
   if  %nullind (fieldname1);
       // field is null
   endif;

   // Set the null indicator for a null-capable field.
   %nullind(fieldname1) = *ON;
   %nullind (fieldname2) = *OFF;
/END-FREE
```

Figure 174. %NULLIND Example

%OCCUR (Set/Get Occurrence of a Data Structure)

```
%OCCUR(dsn-name)
```

%OCCUR gets or sets the current position of a multiple-occurrence data structure.

When this function is evaluated for its value, it returns the current occurrence number of the specified data structure. This is an unsigned numeric value.

When this function is specified on the left-hand side of an EVAL statement, the specified number becomes the current occurrence number. This must be a non-float numeric value with zero decimal places. Exception 00122 is issued if the value is less than 1 or greater than the total number of occurrences.

For more information about multiple-occurrence data structures and the OCCUR operation code, see "OCCUR (Set/Get Occurrence of a Data Structure)" on page 637.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D mds             DS                  OCCURS(10)

 /FREE
   n = %OCCUR(mds);
   // n = 1

   %OCCUR(mds) = 7;

   n = %OCCUR(mds);
   // n = 7
 /END-FREE
```

Figure 175. %OCCUR Example

%OPEN (Return File Open Condition)

```
%OPEN(file_name)
```

%OPEN returns '1' if the specified file is open. A file is considered "open" if it has been opened by the RPG program during initialization or by an OPEN operation, and has not subsequently been closed. If the file is conditioned by an external indicator and the external indicator was off at program initialization, the file is considered closed, and %OPEN returns '0'.

For more information, see "File Operations" on page 363 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * The printer file is opened in the calculation specifications
FQSYSPRT   O    F  132        PRINTER USROPN

 /FREE
    // Open the file if it is not already open
    if not %open (QSYSPRT);
       open QSYSPRT;
    endif;
 /END-FREE
```

Figure 176. %OPEN Example

%PADDR (Get Procedure Address)

```
%PADDR(string)
```

%PADDR returns a value of type procedure pointer. The value is the address of the entry point identified by the argument.

%PADDR may be compared with and assigned to only items of type procedure pointer.

The parameter to %PADDR must be a character or hexadecimal literal or a constant name that represents a character or hexadecimal literal. The entry point name specified by the character string must be found at program bind time and must be in the correct case.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D
D PROC            S               *   PROCPTR
D                                     INZ (%PADDR ('FIRSTPROG'))
D PROC1           S               *   PROCPTR
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C*
C*  The following statement calls procedure 'FIRSTPROG'.
C*
C                   CALLB     PROC
 *----------------------------------------------------------------
C* The following statements call procedure 'NextProg'.
C* This a C procedure and is in mixed case.  Note that
C* the procedure name is case sensitive.
C*
C                   EVAL      PROC1 = %PADDR ('NextProg')
C                   CALLB     PROC1
```

Figure 177. %PADDR Example

%REALLOC (Reallocate Storage)

```
%REALLOC(ptr:num)
```

%REALLOC changes the heap storage pointed to by the first parameter to be the length specified in the second parameter. The newly allocated storage is uninitialized.

The first parameter must be a basing pointer value. The second parameter must be a non-float numeric value with zero decimal places. The length specified must be between 1 and 16776704.

The function returns a pointer to the allocated storage. This may be the same as *ptr* or different.

For more information, see "Memory Management Operations" on page 367.

If the operation cannot complete successfully, exception 00425 or 00426 is issued.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
  // Allocate an area of 200 bytes
  pointer = %ALLOC(200);
  // Change the size of the area to 500 bytes
  pointer = %REALLOC(pointer:500);
/END-FREE
```

Figure 178. %REALLOC Example

%REM (Return Integer Remainder)

```
%REM(n:m)
```

%REM returns the remainder that results from dividing operands **n** by **m**. The two operands must be numeric values with zero decimal positions. If either operand is a packed, zoned, or binary numeric value, the result is packed numeric. If either operand is an integer numeric value, the result is integer. Otherwise, the result is unsigned numeric. Float numeric operands are not allowed. The result has the same sign as the dividend. (See also "%DIV (Return Integer Portion of Quotient)" on page 431.)

%REM and %DIV have the following relationship:

```
%REM(A:B) = A - (%DIV(A:B) * B)
```

If the operands are constants that can fit in 8-byte integer or unsigned fields, constant folding is applied to the built-in function. In this case, the %REM built-in function can be coded in the definition specifications.

For more information, see "Arithmetic Operations" on page 348 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D A               S             10I 0 INZ(123)
D B               S             10I 0 INZ(27)
D DIV             S             10I 0
D REM             S             10I 0
D E               S             10I 0

 /FREE
    DIV = %DIV(A:B);  // DIV is now 4
    REM = %REM(A:B);  // REM is now 15
    E = DIV*B + REM;  // E is now 123
 /END-FREE
```

Figure 179. %DIV and %REM Example

%REPLACE (Replace Character String)

```
%REPLACE(replacement string: source string{:start position :source
length to replace})
```

%REPLACE returns the character string produced by inserting a replacement string into the source string, starting at the start position and replacing the specified number of characters.

The first and second parameter must be of type character, graphic, or UCS-2 and can be in either fixed- or variable-length format. The second parameter must be the same type as the first.

The third parameter represents the starting position, measured in characters, for the replacement string. If it is not specified, the starting position is at the beginning of the source string. The value may range from one to the current length of the source string plus one.

The fourth parameter represents the number of characters in the source string to be replaced. If zero is specified, then the replacement string is inserted before the specified starting position. If the parameter is not specified, the number of characters replaced is the same as the length of the replacement string. The value must be greater than or equal to zero, and less than or equal to the current length of the source string.

The starting position and length may be any numeric value or numeric expression with no decimal positions.

The returned value is varying length if the source string or replacement string are varying length, or if the start position or source length to replace are variables. Otherwise, the result is fixed length.

For more information, see “String Operations” on page 375 or Chapter 25, “Built-In Functions,” on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D var1            S             30A   INZ('Windsor') VARYING
D var2            S             30A   INZ('Ontario') VARYING
D var3            S             30A   INZ('Canada') VARYING
D fixed1          S             15A   INZ('California')
D date            S               D   INZ(D'1997-02-03')
D result          S            100A   VARYING

 /FREE
    result = var1 + ', ' + 'ON';
  // result = 'Windsor, ON'

  // %REPLACE with 2 parameters to replace text at begining of string:
    result = %replace ('Toronto': result);
  // result = 'Toronto, ON'

  // %REPLACE with 3 parameters to replace text at specified position:
    result = %replace (var3: result: %scan(',': result) + 2);
  // result = 'Toronto, Canada'

  // %REPLACE with 4 parameters to insert text:
    result = %replace (', ' + var2: result: %scan (',': result): 0);
  // result = 'Toronto, Ontario, Canada'

  // %REPLACE with 4 parameters to replace strings with different length
    result = %replace ('Scarborough': result:
   1: %scan (',': result) - 1);
  // result = 'Scarborough, Ontario, Canada'

  // %REPLACE with 4 parameters to delete text:
    result = %replace ('': result: 1: %scan (',': result) + 1);
  // result = 'Ontario, Canada'

  // %REPLACE with 4 parameters to add text to the end of the string:
    result = %replace (', ' + %char(date): result:
   %len (result) + 1: 0);
  // result = 'Ontario, Canada, 1997-02-03'

  // %REPLACE with 3 parameters to replace fixed-length text at
  // specified position:  (fixed1 has fixed-length of 15 chars)
    result = %replace (fixed1: result: %scan (',': result) + 2);
  // result = 'Ontario, California     -03'

  // %REPLACE with 4 parameters to prefix text at beginning:
    result = %replace ('Somewhere else: ': result: 1: 0);
  // result = 'Somewhere else: Ontario, California     -03'
 /END-FREE
```

Figure 180. %REPLACE Example

%SCAN (Scan for Characters)

```
%SCAN(search argument : source string {: start})
```

%SCAN returns the first position of the search argument in the source string, or 0 if it was not found. If the start position is specified, the search begins at the starting position. The result is always the position in the source string even if the starting position is specified. The starting position defaults to 1.

The first parameter must be of type character, graphic, or UCS-2. The second parameter must be the same type as the first parameter. The third parameter, if specified, must be numeric with zero decimal positions.

When any parameter is variable in length, the values of the other parameters are checked against the current length, not the maximum length.

The type of the return value is unsigned integer. This built-in function can be used anywhere that an unsigned integer expression is valid.

If the search argument contains trailing blanks, the scan will include those trailing blanks. For example if 'b' represents a blank, %SCAN('12b':'12312b') would return 4. If trailing blanks should not be considered in the scan, use %TRIMR on the search argument. For example %SCAN(%TRIMR('12b'):'12312b') would return 1.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

Note: Unlike the SCAN operation code, %SCAN cannot return an array containing all occurrences of the search string and its results cannot be tested using the %FOUND built-in function.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D source          S             15A   inz ('Dr. Doolittle')
D pos             S              5U 0
D posTrim         S              5U 0
D posVar          S              5U 0
D srchFld         S             10A
D srchFldVar      S             10A   varying
 /FREE
     pos = %scan ('oo' : source);
     // After the EVAL, pos = 6 because 'oo' begins at position 6 in
     // 'Dr. Doolittle'.
     pos = %scan ('D' : source : 2);
     // After the EVAL, pos = 5 because the first 'D' found starting from
     // position 2 is in position 5.
     pos = %scan ('abc' : source);
     // After the EVAL, pos = 0 because 'abc' is not found in
     // 'Dr. Doolittle'.
     pos = %scan ('Dr.' : source : 2);
     // After the EVAL, pos = 0 because 'Dr.' is not found in
     // 'Dr. Doolittle', if the search starts at position 2.
     srchFld = 'Dr.';
     srchFldVar = 'Dr.';
     pos = %scan (srchFld : source);
     posTrim = %scan (%trimr(srchFld) : source);
     posVar = %scan (srchFldVar : source);
     // After the EVAL, pos = 0 because srchFld is a 10-byte field, so
     // the search argument is 'Dr.' followed by seven blanks.  However,
     // posTrim and posVar are both 1, since the %TRIMR and srchFldVar
     // scans both use a 3-byte search argument 'Dr.', no trailing blanks.
 /END-FREE
```

Figure 181. %SCAN Example

%SECONDS (Number of Seconds)

```
%SECONDS(number)
```

%SECONDS converts a number into a duration that can be added to a time or timestamp value.

%SECONDS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a time or timestamp. The result is a time or timestamp value with the appropriate number of seconds added or subtracted. For a time, the resulting value is in *ISO format.

For an example of date and time arithmetic operations, see Figure 173 on page 458.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

%SETATR (Set Attribute)

```
%SETATR(window_name:part_name:attribute_name)
```

%SETATR sets the attribute value of a part on a window. Both the first and second parameters can be %WINDOW or %PART.

For an alternative form of accessing part attributes, see "Qualified GUI Part Attribute Access" on page 379.

Notes:

1. The %SETATR built-in function does not affect the corresponding program fields for parts. To ensure that the attribute value and the value in the program field are the same, use the program field when setting the attribute value. This applies to attributes that have program fields mapped to them, such as entry fields with the TEXT attribute.
2. The %SETATR built-in function does not support 1-byte and 8-byte signed and unsigned integer values, and unicode values.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
     ENT0000B = *BLANKS;
     %setatr('inventory':'ent0000b':'text') = ENT0000B;
/END-FREE
```

Figure 182. %SETATR Example

%SIZE (Size of Constant or Field)

```
%SIZE(variable)
%SIZE(literal)
%SIZE(array{:*ALL})
%SIZE(table{:*ALL})
%SIZE(multiple occurrence data structure{:*ALL})
```

%SIZE returns the number of bytes occupied by the constant or field. The argument may be a literal, a named constant, a data structure, a data structure subfield, a field, an array or a table name. It cannot contain an expression, but some constant-valued built-in functions and constant expressions may be accepted. The value returned is in unsigned integer format (type U).

For a graphic literal, the size is the number of bytes occupied by the graphic characters. For a hexadecimal or UCS-2 literal, the size returned is half the number of hexadecimal digits in the literal.

For variable-length fields, %SIZE returns the total number of bytes occupied by the field (two bytes longer than the declared maximum length).

If the argument is an array name, table name, or multiple occurrence data structure name, the value returned is the size of one element or occurrence. If *ALL is specified as the second parameter for %SIZE, the value returned is the storage taken up by all elements or occurrences. For a multiple occurrence data structure containing pointer subfields, the size may be greater than the size of one occurrence times the number of occurrences. This is possible because the system requires that pointers be placed in storage at addresses evenly divisible by 16. This means that the length of each occurrence may have to be increased enough to make the length an exact multiple of 16 so that the pointer subfields will be positioned correctly in storage for every occurrence.

%SIZE may be specified anywhere that a numeric constant is allowed on the definition specification and in an expression in the extended-factor 2 field of the calculation specification.

For more information, see “Size Operations” on page 375 or Chapter 25, “Built-In Functions,” on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D arr1            S             10    DIM(4)
D table1          S              5    DIM(20)
D field1          S             10
D field2          S              9B 0
D field3          S              5P 2
D num             S              5P 0
D mds             DS            20    occurs(10)
D mds_size        C                   const (%size (mds: *all))
D mds_ptr         DS            20    OCCURS(10)
D   pointer                      *
D vCity           S             40A   VARYING INZ('North York')
D fCity           S             40A           INZ('North York')

 /FREE
     num = %SIZE(field1);        //  10
     num = %SIZE('HH');          //   2
     num = %SIZE(123.4);         //   4
     num = %SIZE(-03.00);        //   4
     num = %SIZE(arr1);          //  10
     num = %SIZE(arr1:*ALL);     //  40
     num = %SIZE(table1);        //   5
     num = %SIZE(table1:*ALL);   // 100
     num = %SIZE(mds);           //  20
     num = %SIZE(mds:*ALL);      // 200
     num = %SIZE(mds_ptr);       //  20
     num = %SIZE(mds_ptr:*ALL);  // 320
     num = %SIZE(field2);        //   4
     num = %SIZE(field3);        //   3
     n1 = %SIZE(vCity);          //  42
     n2 = %SIZE(fCity);          //  40
 /END-FREE
```

Figure 183. %SIZE Example

%SQRT (Square Root of Expression)

```
%SQRT(numeric expression)
```

%SQRT returns the square root of the specified numeric expression. If the operand is of type float, the result is of type float; otherwise, the result is packed decimal numeric. If the parameter has a value less than zero, exception 00101 is issued.

For more information, see "Arithmetic Operations" on page 348 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D n               S             10I 0
D p               S              9P 2
D f               S              4F

 /FREE

   n = %SQRT(239874);
   // n = 489

   p = %SQRT(239874);
   // p = 489.76

   f = %SQRT(239874);
   // f = 489.7693
 /END-FREE
```

Figure 184. %SQRT Example

%STATUS (Return File or Program Status)

```
%STATUS{(file_name)}
```

%STATUS returns the most recent value set for the program or file status. %STATUS is set whenever the program status or any file status changes, usually when an error occurs.

If %STATUS is used without the optional file_name parameter, then it returns the program or file status most recently changed. If a file is specified, the value contained in the INFDS *STATUS field for the specified file is returned. The INFDS does not have to be specified for the file.

%STATUS starts with a return value of 00000 and is reset to 00000 before any operation with an 'E' extender specified begins.

%STATUS is best checked immediately after an operation with the 'E' extender or an error indicator specified, or at the beginning of an INFSR or the *PSSR subroutine.

For more information, see "File Operations" on page 363, "Result Operations" on page 375, or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* The 'E' extender indicates that if an error occurs, the error
* is to be handled as though an error indicator were coded.
* The success of the operation can then be checked using the
* %ERROR built-in function.  The status associated with the error
* can be checked using the %STATUS built-in function.
/FREE
   read(e) InFile;
   if %error;
      exsr CheckError;
   endif;

 //---------------------------------------------------------------------
 // CheckError: Subroutine to process a file I/O error
 //---------------------------------------------------------------------
   begsr CheckError;
      select;
      when  %status < 01000;

         // No error occurred
      when  %status = 01211;
         // Attempted to read a file that was not open
         exsr InternalError;

      when  %status = 01331;
         // The wait time was exceeded for a READ operation
         exsr TimeOut;

      when  %status = 01251;
         // Permanent I/O error
         exsr PermError;

      other;
         // Some other error occurred
         exsr FileError;
      endsl;
   endsr;
/END-FREE
```

Figure 185. %STATUS and %ERROR with 'E' Extender

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D Zero            S              5P 0 INZ(0)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * %STATUS starts with a value of 0
 *
 * The following SCAN operation will cause a branch to the *PSSR
 * because the start position has a value of 0.
C     'A'           SCAN      'ABC':Zero    Pos
C     BAD_SCAN      TAG
 * The following EXFMT operation has an 'E' extender, so %STATUS will
 * be set to 0 before the operation begins.  Therefore, it is
 * valid to check %STATUS after the operation.
 * Since the 'E' extender was coded, %ERROR can also be used to
 * check if an error occurred.
C                   READ(E)  REC1
C                   IF        %ERROR
C                   SELECT
C                   WHEN      %STATUS = 01211
C ...
C                   WHEN      %STATUS = 01299
C ...
 * The following scan operation has an error indicator.  %STATUS will
 * not be set to 0 before the operation begins, but %STATUS can be
 * reasonably checked if the error indicator is on.
C     'A'           SCAN      'ABC':Zero    Pos                   10
C                   IF        *IN10 AND %STATUS = 00100
C ...

 * The following scan operation does not produce an error.
 * Since there is no 'E' extender %STATUS will not be set to 0,
 * so it would return a value of 00100 from the previous error.
 * Therefore, it is unwise to use %STATUS after an operation that
 * does not have an error indicator or the 'E' extender coded since
 * you cannot be sure that the value pertains to the previous
 * operation.
C     'A'           SCAN      'ABC'         Pos
C ...
C     *PSSR         BEGSR
 * %STATUS can be used in the *PSSR since an error must have occurred.
C                   IF        %STATUS = 00100
C                   GOTO      BAD_SCAN
C ...
```

*Figure 186. %STATUS and %ERROR with 'E' Extender, Error Indicator and *PSSR*

%STR (Get or Store Null-Terminated String)

```
%STR(basing pointer{: max-length})(right-hand-side)
%STR(basing pointer : max-length)(left-hand-side)
```

%STR is used to create or use null-terminated character` strings, which are very commonly used in C and C++ applications.

The first parameter must be a basing-pointer value. (Any basing pointer expression is valid, such as ″%ADDR(DATA)″ or ″P+1″.) The second parameter, if specified, must be a numeric value with zero decimal positions. If not specified, it defaults to 65535.

The first parameter must point to storage that is at least as long as the length given by the second parameter.

Error conditions:
1. If the length parameter is not between 1 and 65535, an error will occur with status 00100.
2. If the pointer is not set, an error will occur with status code 00222.
3. If the storage addressed by the pointer is shorter than indicated by the length parameter, either
   a. An error will occur with status code 00222
   b. Data corruption will occur

For more information, see “String Operations” on page 375 or Chapter 25, “Built-In Functions,” on page 399.

%STR Used to Get Null-Terminated String

When used on the right-hand side of an expression, this function returns the data pointed to by the first parameter up to but not including the first null character (x’00’) found within the length specified. This built-in function can be used anywhere that a character expression is valid. No error will be given at run time if the null terminator is not found within the length specified. In this case, the length of the resulting value is the same as the length specified.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D String1         S               *
D Fld1            S             10A

 /FREE
    Fld1 = '<' + %str(String1) + '>';
  // Assuming that String1 points to '123¬' where '¬' represents the
  // null character, after the EVAL, Fld1 = '<123>     '.
 /END-FREE
```

Figure 187. %STR (right-hand-side) Example 1

The following is an example of %STR with the second parameter specified.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D String1         S               *
D Fld1            S             10A

 /FREE
     Fld1 = '<' + %str(String1 : 2) + '>';
  // Assuming that String1 points to '123¬' where '¬' represents the
  // null character, after the EVAL, Fld1 = '<12>      '.
  // Since the maximum length read by the operation was 2, the '3' and
  // the '¬' were not considered.
 /END-FREE
```

Figure 188. %STR (right-hand-side) Example 2

In this example, the null-terminator is found within the specified maximum length.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D String1         S               *
D Fld1            S             10A

 /FREE
    Fld1 = '<' + %str(String1 : 5) + '>';
    // Assuming that String1 points to '123¬' where '¬' represents the
    // null character, after the EVAL, Fld1 = '<123>     '.
    // Since the maximum length read by the operation was 5, the
    // null-terminator in position 4 was found so all the data up to
    // the null-terminator was used.
 /END-FREE
```

Figure 189. %STR (right-hand-side) Example 3

%STR Used to Store Null-Terminated String

When used on the left-hand side of an expression, %STR(ptr:length) assigns the value of the right-hand side of the expression to the storage pointed at by the pointer, adding a null-terminating byte at the end. The maximum length that can be specified is 65535. This means that at most 65534 bytes of the right-hand side can be used, since 1 byte must be reserved for the null-terminator at the end.

The length indicates the amount of storage that the pointer points to. This length should be greater than the maximum length the right-hand side will have. The pointer must be set to point to storage at least as long as the length parameter. If the length of the right-hand side of the expression is longer than the specified length, the right-hand side value is truncated.

Note: Data corruption will occur if both of the following are true:

1. The length parameter is greater than the actual length of data addressed by the pointer.
2. The length of the right-hand side is greater than or equal to the actual length of data addressed by the pointer.

If you are dynamically allocating storage for use by %STR, you must keep track of the length that you have allocated.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D String1         S               *
D Fld1            S             10A

 /FREE
     %str(String1: 25)= 'abcdef';
     // The storage pointed at by String1 now contains 'abcdef¬'
     // Bytes 8-25 following the null-terminator are unchanged.

     %str (String1: 4) = 'abcdef';
     // The storage pointed at by String1 now contains 'abc¬'
 /END-FREE
```

Figure 190. %STR (left-hand-side) Examples

%SUBARR (Set/Get Portion of an Array)

```
%SUBARR(array:start-index{:number-of-elements})
```

Built-in function %SUBARR returns a section of the specified array starting at *start-index*. The number of elements returned is specified by the optional *number-of-elements* parameter. If not specified, the *number-of-elements* defaults to the remainder of the array.

The first parameter of %SUBARR must be an array. That is, a standalone field, data structure, or subfield defined as an array. The first parameter must not be a table name or procedure call.

The *start-index* parameter must be a numeric value with zero decimal positions. A float numeric value is not allowed. The value must be greater than or equal to 1 and less than or equal to the number of elements of the array.

The optional *number-of-elements* parameter must be a numeric value with zero decimal positions. A float numeric value is not allowed. The value must be greater than or equal to 1 and less than or equal to the number of elements remaining in the array after applying the *start-index* value.

Generally, %SUBARR is valid in any expression where an unindexed array is allowed. However, %SUBARR cannot be used in the following places:

- as the array argument of built-in function %LOOKUPxx
- as a parameter passed by reference

%SUBARR may be used in the following ways:

- On the left-hand side of an assignment using EVAL or EVALR. This changes the specified elements in the specified array.
- Within the expression on the right-hand side of an assignment using EVAL or EVALR where the target of the assignment is an array. This uses the values of the specified elements of the array. The array elements are used directly; a temporary copy of the sub-array is not made.
- In Extended Factor 2 of the SORTA operation.
- In Extended Factor 2 of the RETURN operation.
- Passed by VALUE or by read-only reference (CONST keyword) when the corresponding parameter is defined as an array.
- As the parameter of the %XFOOT builtin function.

For more information, see “Array Operations” on page 351 or Chapter 25, “Built-In Functions,” on page 399.

```
D a               s             10i 0 dim(5)
D b               s             10i 0 dim(15)
D resultArr       s             10i 0 dim(20)
D sum             s             20i 0
 /free
     a(1)=9;
     a(2)=5;
     a(3)=16;
     a(4)=13;
     a(5)=3;
     // Copy part of an array to another array:
     resultArr = %subarr(a:4:n);
          // this is equivalent to:
          //   resultArr(1) = a(4)
          //   resultArr(2) = a(5)
          //   ...
          //   resultArr(n) = a(4 + n - 1)

     // Copy part of an array to part of another array:
     %subarr(b:3:n) = %subarr(a:m:n);
     // Specifying the array from the start element to the end of the array
     // B has 15 elements and A has 5 elements.  Starting from element 2
     // in array A means that only 4 elements will be copied to array B.
     // The remaining elements in B will not be changed.
      b = %subarr(a : 2);

     // Sort a subset of an array:
     sorta %subarr(a:1:4);
          // Now, A=(5 9 13 16 3);
          // Since only 4 elements were sorted, the fifth element
          // is out of order.
          // Using %SUBARR in an implicit array indexing assignment
     resultArr = b + %subarr(a:2:3)
     // this is equivalent to:
     //   resultArr(1) = b(1) + a(2)
     //   resultArr(2) = b(2) + a(3)
     //   resultArr(3) = b(3) + a(4)

     // Using %SUBARR nested within an expression
     resultArr = %trim(%subst(%subarr(stringArr:i):j));
          // this is equivalent to:
          //   resultArr(1) = %trim(%subst(stringArr(i+0):j))
          //  resultArr(2) = %trim(%subst(stringArr(i+1):j))
          //   resultArr(3) = %trim(%subst(stringArr(i+2):j))

     // Sum a subset of an array
     sum = %xfoot (%subarr(a:2:3));
          // Now sum = 9 + 13 + 16 = 38
```

Figure 191. Using %SUBARR

```
// Using %SUBARR with dynamically allocated arrays
D dynArrInfo      ds                  qualified
D   numAlloc                    10i 0 inz(0)
D   current                     10i 0 inz(0)
D   p                             *
D dynArr          s              5a   dim(32767) based(dynArrInfo.p)
D otherArray      s              3a   dim(10) inz('xy')
 /free
     // Start the array with an allocation of five elements,
     // and with two current elements
     dynArrInfo.numAlloc = 5;
     dynArrInfo.p = %alloc(%size(dynArr) *
                    dynarrInfo.numAlloc);
     dynArrInfo.current = 2;
     // Initialize to blanks
     %subarr(dynArr : 1 : dynarrInfo.current) = *blank;

     // Set the two elements to some values
     dynArr(1) = 'Dog';  dynArr(2) = 'Cat';

     // Sort the two elements
     sorta %subarr(dynArr : 1 : dynarrInfo.current);
          // dynArr(1) = 'Cat'
          // dynArr(2) = 'Dog'

     // Assign another array to the two elements
     otherArray(1) = 'ab';
     otherArray(2) = 'cd';
     otherArray(3) = 'ef';
     %subarr(dynArr : 1 : dynarrInfo.current) = otherArray;
          // dynArr(1) = 'ab'
          // dynArr(2) = 'cd'

     // Changing the size of the array
     oldElems = dynArrInfo.current;
     dynArrInfo.current = 7;
     if (dynArrInfo.current > dynArrInfo.alloc);
          dynArrInfo.p = %realloc (dynArrInfo.p : dynArrInfo.current);
          dynArrInfo.numAlloc = dynArrInfo.current;
     endif;
     if (oldElems < dynArrInfo.current);
          // Initialize new elements to blanks
          %subarr(dynArr : oldElems + 1 : dynArrInfo.current - oldElems);
     endif;
```

Figure 192. Using %SUBARR with dynamically allocated arrays

CAUTION:
It is valid to use %SUBARR to assign part of an array to another part of the same array. However, if the source part of the array overlaps the target part of the array, unpredictable results can occur.

For more information, see Chapter 25, "Built-In Functions," on page 399.

%SUBDT (Extract a Portion of a Date, Time, or Timestamp)

```
%SUBDT(value:*MSECONDS|*SECONDS|*MINUTES|*HOURS|*DAYS|*MONTHS|*YEARS)
%SUBDT(value:*MS|*S|*MN|*H|*D|*M|*Y)
```

%SUBDT extracts a portion of the information in a date, time, or timestamp value. It returns an unsigned numeric value.

The first parameter is the date, time, or timestamp value.

The second parameter is the portion that you want to extract. The following values are valid:

- For a date: *DAYS, *MONTHS, and *YEARS
- For a time: *SECONDS, *MINUTES, and *HOURS
- For a timestamp: *MSECONDS, *SECONDS, *MINUTES, *HOURS, *DAYS, *MONTHS, and *YEARS

For this function, *DAYS always refers to the day of the month not the day of the year (even if you are using a Julian date format). For example, the day portion of February 10 is 10 not 41.

This function always returns a 4-digit year, even if the date format has a 2-digit year.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

    date = d'1999-02-17';
    time = t'01.23.45';

    num = %subdt(date:*YEARS);
    // num = 1999

    num = %subdt(time:*MN);
    // num = 23
/END-FREE
```

Figure 193. %SUBDT Example

%SUBST (Get Substring)

```
%SUBST(string:start{:length})
```

%SUBST returns a portion of argument string. It may also be used as the result of an assignment with the EVAL operation code.

The start parameter represents the starting position of the substring.

The length parameter represents the length of the substring. If it is not specified, the length is the length of the string parameter less the start value plus one.

The string must be character, graphic, or UCS-2 data. Starting position and length may be any numeric value or numeric expression with zero decimal positions. The starting position must be greater than zero. The length may be greater than or equal to zero.

When the string parameter is varying length, the values of the other parameters are checked against the current length, not the maximum length.

When specified as a parameter for a definition specification keyword, the parameters must be literals or named constants representing literals. When specified on a free-form calculation specification, the parameters may be any expression.

For more information, see “String Operations” on page 375 or Chapter 25, “Built-In Functions,” on page 399.

%SUBST Used for its Value

%SUBST returns a substring from the contents of the specified string. The string may be any character, graphic, or UCS-2 field or expression. Unindexed arrays are allowed for string, start, and length. The substring begins at the specified starting position in the string and continues for the length specified. If length is not specified then the substring continues to the end of the string. For example:

```
The value of  %subst('Hello World': 5+2) is  'World'
The value of  %subst('Hello World':5+2:10-7) is 'Wor'
The value of  %subst('abcd' + 'efgh':4:3) is 'def'
```

For graphic or UCS-2 characters the start position and length is consistent with the 2-byte character length (position 3 is the third 2-byte character and length 3 represents 3 2-byte characters to be operated on).

%SUBST Used as the Result of an Assignment

When used as the result of an assignment this built-in function refers to certain positions of the argument string. Unindexed arrays are not allowed for start and length.

The result begins at the specified starting position in the variable and continues for the length specified. If length is not specified or it refers to characters beyond the end of the string then the string is referenced to its end.

When %SUBST is used as the result of an assignment, the first parameter must refer to a storage location. That is, the first parameter of the %SUBST operation must be one of the following.

- Field
- Data Structure
- Data Structure Subfield

- Array Name
- Array Element
- Table Element

Any valid expressions are permitted for the the second and third parameters of %SUBST when it appears as the result of an assignment with an EVAL operation.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C*
C* In this example, CITY contains 'Toronto, Ontario'
C* %SUBST returns the value 'Ontario'.
C*
C     ' '           SCAN      CITY          C
C                   IF        %SUBST(CITY:C+1) = 'Ontario'
C                   EVAL      CITYCNT = CITYCNT+1
C                   ENDIF
C*
C* Before the EVAL, A has the value 'abcdefghijklmno'.
C* After the EVAL A has the value 'ab****ghijklmno'
C*
C                   EVAL      %SUBST(A:3:4) = '****'
```

Figure 194. %SUBST Example

%THIS (Return Class Instance for Native Method)

```
%THIS
```

%THIS returns an Object value that contains a reference to the class instance on whose behalf the native method is being called. %THIS is valid only in non-static native methods. This built-in gives non-static native methods access to the class instance.

A non-static native method works on a specific instance of its class. This object is actually passed as a parameter to the native method by Java, but it does not appear in the prototype or procedure interface for the native method. In a Java method, the object instance is referred to by the Java reserved word ***this***. In an RPG native method, the object instance is referred to by the %THIS builtin function.

```
 * Method "vacationDays" is a method in the class 'Employee'
D vacationDays    PR            10I 0 EXTPROC(*JAVA
D                                     : 'Employee'
D                                     : 'vacationDays')

 * Method "getId" is another method in the class 'Employee'
D getId           PR            10I 0 EXTPROC(*JAVA
D                                     : 'Employee'
D                                     : 'getId')
...
 * "vacationDays" is an RPG native method.  Since the STATIC keyword
 * is not used, it is an instance method.
P vacationDays    B
D vacationDays    PI            10I 0

D id_num          S             10I 0

 * Another Employee method must be called to get the Employee's
 * id-number.  This method requires an Object of class Employee.
 * We use %THIS as the Object parameter, to get the id-number for
 * the object that our native method "vacationDays" is working on.
C                   eval      id_num = getId(%THIS)
C     id_num        chain     EMPFILE
C                   if        %found
C                   return    VACDAYS
C                   else
C                   return    -1
C                   endif

P vacationDays    E
```

Figure 195. %THIS Example

%TIME (Convert to Time)

```
%TIME{(expression{:time-format})}
```

%TIME converts the value of the expression from character, numeric, or timestamp data to type time. The converted value remains unchanged, but is returned as a time.

The first parameter is the value to be converted. If you do not specify a value, %TIME returns the current system time.

The second parameter is the time format for numeric or character input. Regardless of the input format, the output is returned in *ISO format.

For information on the input formats that can be used, see "Time Data" on page 135. If the time format is not specified for numeric or character input, the default value is either the format specified on the TIMFMT control-specification keyword or *ISO. For more information, see "TIMFMT(fmt{separator})" on page 235.

If the first parameter is a timestamp, do not specify the second parameter. The system knows the format of the input in this case.

For more information, see "Information Operations" on page 366 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

   string = '12:34 PM';
   time =  %time(string:*USA);
   // time = t'12.34.00'
/END-FREE
```

Figure 196. %TIME Example

%TIMESTAMP (Convert to Timestamp)

```
%TIMESTAMP{(expression{:*ISO|*ISO0})}
```

%TIMESTAMP converts the value of the expression from character, numeric, or date data to type timestamp. The converted value is returned as a timestamp.

The first parameter is the value to be converted. If you do not specify a value, %TIMESTAMP returns the current system timestamp.

The second parameter is the timestamp format for character input. Regardless of the input format, the output is returned in *ISO format. You can specify either *ISO (the default) or *ISO0. For more information, see "Timestamp Data" on page 137.

If the first parameter is numeric, you do not need to specify the second parameter. The only allowed value is *ISO (the default).

If the first parameter is a date, do not specify the second parameter. The system converts the date from its current format to *ISO format and adds `00.00.00.0000`.

For more information, see "Information Operations" on page 366 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

  string = '1960-09-29-12.34.56.000000';
  timest =  %timestamp(string);
  // timest now contains t'1960-09-29-12.34.56.000000'
/END-FREE
```

Figure 197. %TIMESTAMP Example

%TLOOKUPxx (Look Up a Table Element)

```
%TLOOKUP(arg : search-table {: alt-table})
%TLOOKUPLT(arg : search-table {: alt-table})
%TLOOKUPGE(arg : search-table {: alt-table})
%TLOOKUPGT(arg : search-table {: alt-table})
%TLOOKUPLE(arg : search-table {: alt-table})
```

The following functions search *search-table* for a value that matches *arg* as follows:

%TLOOKUP An exact match.

%TLOOKUPLT
The value that is closest to *arg* but less than *arg*.

%TLOOKUPLE
An exact match, or the value that is closest to *arg* but less than *arg*.

%TLOOKUPGT
The value that is closest to *arg* but greater than *arg*.

%TLOOKUPGE
An exact match, or the value that is closest to *arg* but greater than *arg*.

If a value meets the specified condition, the current table element for the search table is set to the element that satisfies the condition, the current table element for the alternate table is set to the same element, and the function returns the value *ON.

If no value matches the specified condition, *OFF is returned.

The first two parameters can have any type but must have the same type. They do not need to have the same length or number of decimal positions.

Built-in functions %FOUND and %EQUAL are not set following a %LOOKUP operation.

Note: Unlike the LOOKUP operation code, %TLOOKUP applies only to tables. To look up a value in an array, use the %LOOKUP built-in function.

The %TLOOKUPxx builtin functions use a binary search for sequenced tables (tables that have the ASCEND or DESCEND keyword specified). See "Sequenced arrays that are not in the correct sequence" on page 456.

For more information, see "Array Operations" on page 351 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
  *IN01 = %TLOOKUP('Paris':tab1);
  IF %TLOOKUP('Thunder Bay':tab1:tab2);
  // code to handle Thunder Bay
  ENDIF;
/END-FREE
```

Figure 198. %TLOOKUPxx Example

%TRIM (Trim Characters at Edges)

```
%TRIM(string {: characters to trim})
```

%TRIM with only one parameter returns the given string with any leading and trailing blanks removed.

%TRIM with two parameters returns the given string with any leading and trailing characters that are in the *characters to trim parameter* removed.

The string can be character, graphic, or UCS-2 data.

If the *characters to trim* parameter is specified, it must be the same type as the *string* parameter.

When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Note: Specifying %TRIM with two parameters is not supported for parameters of Definition keywords.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D Location        S             16A
D FirstName       S             10A   inz ('   Chris ')
D LastName        S             10A   inz ('   Smith ')
D Name            S             20A

 * LOCATION will have the value 'Toronto, Ontario'.
 /FREE
     Location = %trim ('  Toronto, Ontario  ');

  // Name will have the value 'Chris Smith!        '.
   Name = %trim (FirstName) + ' ' + %trim (LastName) + '!';
 /END-FREE
```

Figure 199. %TRIM Example

```
D edited          S             20A   INZ('$******5.27***      ')
D trimmed         S             20A   varying
D numeric         S             15P 3
 /FREE

     // Trim '$' and '*' from the edited numeric value
     // Note: blanks will not be trimmed, since a blank
     // is not specified in the 'characters to trim' parameter

     trimmed = %trim(edited : '$*');
     // trimmed is now '5.27***      '

     // Trim '$' and '*' and blank from the edited numeric value

     trimmed = %trim(edited : '$* ');
     // trimmed is now '5.27'

     // Get the numeric value from the edited value

     numeric = %dec(%trim(edited : '$* ') : 31 : 9);
     // numeric is now 5.27
```

Figure 200. Trimming characters other than blank

%TRIML (Trim Leading Characters)

```
%TRIML(string {: characters to trim})
```

%TRIML with only one parameter returns the given string with any leading blanks removed.

%TRIML with two parameters returns the given string with any leading characters that are in the *characters to trim parameter* removed.

The string can be character, graphic, or UCS-2 data.

If the *characters to trim* parameter is specified, it must be the same type as the *string* parameter.

When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Note: Specifying %TRIML with two parameters is not supported for parameters of Definition keywords.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* LOCATION will have the value 'Toronto, Ontario  '.
/FREE
    // Trimming blanks
    Location = %triml('  Toronto, Ontario  ');
    // LOCATION now has the value 'Toronto, Ontario  '.

    // Trimming other characters

    trimmed = %triml('$******5.27***        ' : '$* ');
                      // trimmed is now '5.27***        '
```

Figure 201. %TRIML Example

%TRIMR (Trim Trailing Characters)

```
%TRIMR(string {: characters to trim})
```

%TRIMR with only one parameter returns the given string with any trailing blanks removed.

%TRIMR with two parameters returns the given string with any trailing characters that are in the *characters to trim parameter* removed.

The string can be character, graphic, or UCS-2 data.

If the *characters to trim* parameter is specified, it must be the same type as the *string* parameter.

When specified as a parameter for a definition specification keyword, the string parameter must be a constant.

Note: Specifying %TRIMR with two parameters is not supported for parameters of Definition keywords.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D Location        S             16A   varying
D FirstName       S             10A   inz ('Chris')
D LastName        S             10A   inz ('Smith')
D Name            S             20A   varying

 * LOCATION will have the value '  Toronto, Ontario'.
 /FREE
     Location = %trim ('  Toronto, Ontario  ');

     // Name will have the value 'Chris Smith:'.
     Name = %trimr (FirstName) + ' ' + %trimr (LastName) + ':';
 /END-FREE
```

Figure 202. %TRIMR Example

```
string = '(' + %trimr('$******5.27***      ' : '$*') + ')';
// string is now '($******5.27***      )'
//
// Nothing has been trimmed from the right-hand side because
// the right-most character is a blank, and a blank does not
// appear in the 'characters to trim' parameter

string = '(' + %trimr('$******5.27***      ' : '$ *') + ')';
// string is now '($******5.27)'
```

Figure 203. Trimming characters other than blanks

%UCS2 (Convert to UCS-2 Value)

```
%UCS2(char-expr | graph-expr)
```

%UCS2 converts the value of the expression from character or graphic and returns a UCS-2 value. The result is varying length if the parameter is varying length, or if the parameter is single-byte character.

The second parameter, *ccsid*, is optional and indicates the CCSID of the resulting expression. The CCSID defaults to 13488.

If the parameter is a constant, the conversion will be done at compile time.

If the conversion results in substitution characters, a warning message is issued at compile time. At run time, status 00050 is set and no error message is issued.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H CCSID(*UCS2 : 13488)
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D  char           S               5A   INZ('abcde')
D  graph          S               2G   INZ(G'oAABBi')
 * The %UCS2 built-in function is used to initialize a UCS-2 field.
D  ufield         S              10C   INZ(%UCS2('abcdefghij'))
D  ufield2        S               1C   CCSID(61952) INZ(*LOVAL)
D  isLess                         1N
D  proc           PR
D    uparm                        2G   CCSID(13488) CONST
CSRN01Factor1+++++++Opcode&ExtExtended-factor2++++++++++++++++++++++++++++
C                   EVAL      ufield = %UCS2(char) +  %UCS2(graph)
 * ufield now has 7 UCS-2 characters representing
 * 'a.b.c.d.e.AABB' where 'x.' represents the UCS-2 form of 'x'
C                   EVAL      isLess = ufield < %UCS2(ufield2:13488)
 * The result of the %UCS2 built-in function is the value of
 * ufield2, converted from CCSID 61952 to CCSID 13488
 * for the comparison.

C                   EVAL      ufield = ufield2
 * The value of ufield2 is converted from CCSID 61952 to
 * CCSID 13488 and stored in ufield.
 * This conversion is handled implicitly by the compiler.

C                   CALLP     proc(ufield2)
 * The value of ufield2 is converted to CCSID 13488
 * implicitly, as part of passing the parameter by constant reference.
```

Figure 204. %UCS2 Examples

%UNS (Convert to Unsigned Format)

```
%UNS(numeric or character expression)
```

%UNS converts the value of the expression to unsigned format. Any decimal digits are truncated. %UNS can be used to truncate the decimal positions from a float or decimal value allowing it to be used as an array index.

If the parameter is a character expression, the following rules apply:
- The sign is optional. It can only be '+' . It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Blanks are allowed anywhere in the data. For example, ' + 3 ' is a valid parameter.
- Floating point data is not allowed. That is, where the numeric value is followed by E and an exponent, for example '1.2E6'.
- If invalid numeric data is found, an exception occurs with status code 105

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

Figure 205 on page 496 shows an example of the %UNS built-in function.

%UNSH (Convert to Unsigned Format with Half Adjust)

```
%UNSH(numeric or character expression)
```

%UNSH is the same as %UNS except that if the expression is a decimal, float or character value, half adjust is applied to the value of the expression when converting to integer type. No message is issued if half adjust cannot be performed.

For more information, see "Conversion Operations" on page 358 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D p7              s              7p 3 inz (8236.567)
D s9              s              9s 5 inz (23.73442)
D f8              s              8f   inz (173.789)
D c15a            s             15a   inz (' 12345.6789 +')
D c15b            s             15a   inz (' + 5 , 6 7 ')
D result1         s             15p 5
D result2         s             15p 5
D result3         s             15p 5
D array           s              1a   dim (200)
D a               s              1a

 /FREE
 // using numeric parameters
    result1 = %uns (p7) + 0.1234; // "result1" is now 8236.12340
    result2 = %uns (s9);          // "result2" is now   23.00000
    result3 = %unsh (f8);         // "result3" is now  174.00000
 // using character parameters
     result1 = %uns (c15a);        // "result1" is now 12345.0000
     result2 = %unsh (c15b);       // "result2" is now     6.00000
    // %UNS and %UNSH can be used as array indexes
    a = array (%unsh (f8));
 /END-FREE
```

Figure 205. %UNS and %UNSH Example

%XFOOT (Sum Array Expression Elements)

```
%XFOOT(array-expression)
```

%XFOOT results in the sum of all elements of the specified numeric array expression.

The precision of the result is the minimum that can hold the result of adding together all array elements, up to a maximum of 31 digits. The number of decimal places in the result is always the same as the decimal places of the array expression.

For example, if ARR is an array of 500 elements of precision (17,4), the result of %XFOOT(ARR) is (20,4).

For %XFOOT(X) where X has precision (m,n), the following table shows the precision of the result based on the number of elements of X:

Elements of X	Precision of %XFOOT(X)
1	(m,n)
2-10	(m+1,n)
11-100	(m+2,n)
101-1000	(m+3,n)
1001-10000	(m+4,n)
10001-32767	(m+5,n)

Normal rules for array expressions apply. For example, if ARR1 has 10 elements and ARR2 has 20 elements, %XFOOT(ARR1+ARR2) results in the sum of the first 10 elements of ARR1+ARR2.

This built-in function is similar to the XFOOT operation, except that float arrays are summed like all other types, beginning from index 1 on up.

For more information, see “Array Operations” on page 351 or Chapter 25, “Built-In Functions,” on page 399.

%XLATE (Translate)

```
%XLATE(from:to:string{:startpos})
```

%XLATE translates *string* according to the values of *from*, *to*, and *startpos*.

The first parameter contains a list of characters that should be replaced, and the second parameter contains their replacements. For example, if the string contains the third character in *from*, every occurrence of that character is replaced with the third character in *to*.

The third parameter is the string to be translated. The fourth parameter is the starting position for translation. By default, translation starts at position 1.

The first three parameters can be of type character, graphic, or UCS-2. All three must have the same type. The value returned has the same type and length as *string*.

The fourth parameter is a non-float numeric with zero decimal positions.

For more information, see "String Operations" on page 375 or Chapter 25, "Built-In Functions," on page 399.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D up              C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
D lo              C                   'abcdefghijklmnopqrstuvwxyz'
D string          S             10A   inz('rpg dept')

 /FREE

   string = %XLATE(lo:up:'rpg dept');
   // string now contains 'RPG DEPT'

   string = %XLATE(up:lo:'rpg dept':6);
   // string now contains 'RPG Dept'
 /END-FREE
```

Figure 206. %XLATE Example

%YEARS (Number of Years)

```
%YEARS(number)
```

%YEARS converts a number into a duration that can be added to a date or timestamp value.

%YEARS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp. The result is a date or timestamp value with the appropriate number of years added or subtracted. For a date, the resulting value is in *ISO format.

If the left-hand value is February 29 and the resulting year is not a leap year, February 28 is used instead. Adding or subtracting a number of years to a February 29 date may not be reversible. For example, 2000-02-29 + %YEARS(1) - %YEARS(1) is 2000-02-28.

For an example of the %YEARS built-in function, see Figure 173 on page 458.

For more information, see "Date Operations" on page 359 or Chapter 25, "Built-In Functions," on page 399.

Chapter 26. Operation Code Details

The following sections describe each operation code in detail.

ADD (Add)

Free-Form Syntax	(not allowed - use the + or += operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
ADD (H)	Addend	Addend	Sum	+	–	Z

If factor 1 is specified, the ADD operation adds it to factor 2 and places the sum in the result field. If factor 1 is not specified, the contents of factor 2 are added to the result field and the sum is placed in the result field.

Factor 1 and factor 2 must be numeric and can contain one of: an array, array element, constant, field name, literal, subfield, or table name.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The value 1 is added to RECNO.
C                   ADD       1             RECNO
C* The contents of EHWRK are added to CURHRS.
C                   ADD       EHWRK         CURHRS
C* The contents of OVRTM and REGHRS are added together and
C* placed in TOTPAY.
C     OVRTM         ADD       REGHRS        TOTPAY
```

Figure 207. ADD Operation

ADDDUR (Add Duration)

Free-Form Syntax	(not allowed - use the + or += operators with duration functions such as %YEARS and %MONTHS)

Code	Factor 1	Factor 2	Result Field	Indicators		
ADDDUR (E)	Date/Time	Duration:Duration Code	Date/Time	_	ER	_

The ADDDUR operation adds the duration specified in factor 2 to a date or time and places the resulting Date, Time or Timestamp in the result field.

If factor 1 is specified, it must contain a Date, Time or Timestamp field, subfield, array, array element, literal, or constant.

If factor 1 contains a field name, array or array element then its data type must be the same data type as the field specified in the result field. If factor 1 is not specified, the duration is added to the field specified in the result field.

Factor 2 must contain two subfactors. The first is a duration and must be a numeric field, array element, or constant with zero decimal positions. If the duration is negative, then it is subtracted from the date. The second subfactor must be a valid duration code indicating the type of duration. The duration code must be consistent with the result field data type. A year, month, or day can be added to a date field. A minute duration cannot be added to a date field. “Date Operations” on page 359 describes the duration codes.

The result field must be a date, time or timestamp data type field, array, or array element. If Factor 1 is blank, the duration is added to the value in the result field. If the result field is an array, the value in factor 2 is added to each element of the array. If the result field is a time field, the result will always be a valid time. For example, adding 59 minutes to 23:59:59 would give 24:58:59. Since this time is not valid, the compiler adjusts it to 00:59:59.

When adding a duration in months to a date, the general rule is that the month portion is increased by the number of months in the duration, and the day portion is unchanged. The exception to this is when the resulting day portion would exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month end date. The following examples (which assume a *YMD format) illustrate this point.

```
'98/05/30' ADDDUR 1:*MONTH results in '98/06/30'
```

The resulting month portion has been increased by 1; the day portion is unchanged.

```
'98/05/31' ADDDUR 1:*MONTH results in '98/06/30'
```

The resulting month portion has been increased by 1; the resulting day portion has been adjusted because June has only 30 days.

Similar results occur when adding a year duration. For example, adding one year to ’92/02/29’ results in ’93/02/28’, an adjusted value since the resulting year is not a leap year.

For more information, see “Memory Management Operations” on page 367.

An error situation arises when one of the following occurs:
- The value of the Date, Time, or Timestamp field in factor 1 is invalid
- Factor 1 is blank and the value of the result field before the operation is invalid
- Overflow or underflow occurred (that is, the resulting value is greater than *HIVAL or less than *LOVAL).

In an error situation,
- An error (status code 112 or 113) is signalled.
- The error indicator (columns 73-74) — if specified — is set on, or the %ERROR built-in function — if the 'E' extender is specified — is set to return '1'.
- The value of the result field remains unchanged.

To handle exceptions with program status codes 112 or 113, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

The system places a 15 digit limit on durations. Adding a duration with more than 15 significant digits causes errors or truncation. This can be avoided by limiting the first subfactor in Factor 2 to 15 digits.

For more information, see "Date Operations" on page 359.

ADDDUR (Add Duration)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H TIMFMT(*USA) DATFMT(*MDY&)
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D*
DDateconst        C                   CONST(D'12 31 92')
D*
D* Define a Date field and initialize
D*
DLoandate         S               D   DATFMT(*EUR) INZ(D'12 31 92')
DDuedate          S               D   DATFMT(*ISO)
Dtimestamp        S               Z
Danswer           S               T

CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....

C* Determine a DUEDATE which is xx years, yy months, zz days later
C* than LOANDATE.
C     LOANDATE      ADDDUR    XX:*YEARS     DUEDATE
C                   ADDDUR    YY:*MONTHS    DUEDATE
C                   ADDDUR    ZZ:*DAYS      DUEDATE
C* Determine the date 23 days later
C*
C                   ADDDUR    23:*D         DUEDATE

C* Add a 1234 microseconds to a timestamp
C*
C                   ADDDUR    1234:*MS      timestamp

C* Add 12 HRS and 16 minutes to midnight
C*
C     T'00:00 am'   ADDDUR    12:*Hours     answer
C                   ADDDUR    16:*Minutes   answer

C* Subtract 30 days from a loan due date
C*
C                   ADDDUR    -30:*D        LOANDUE
```

Figure 208. ADDDUR Operation

ALLOC (Allocate Storage)

Free-Form Syntax	(not allowed - use the %ALLOC built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
ALLOC (E)		Length	Pointer	_	ER	_

The ALLOC operation allocates storage in the default heap of the length specified in factor 2. The result field pointer is set to point to the new heap storage. The storage is uninitialized.

Factor 2 must be a numeric with zero decimal positions. It can be a literal, constant, standalone field, subfield, table name or array element. The value must be between 1 and 16776704. If the value is out of range at runtime, an error will occur with status 00425. If the storage could not be allocated, an error will occur with status 426. If these errors occur, the result field pointer remains unchanged.

The result field must be a basing pointer scalar variable (a standalone field, data structure subfield, table name, or array element).

To handle exceptions with program status codes 425 or 426, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "Memory Management Operations" on page 367.

```
D Ptr1            S               *
D Ptr2            S               *

C                   ALLOC     7                 Ptr1
 * Now Ptr1 points to 7 bytes of storage

C                   ALLOC (E) 12345678          Ptr2

 * This is a large amount of storage, and sometimes it may
 * be unavailable.  If the storage could not be allocated,
 * %ERROR will return '1', the status is set to 00426, and
 * %STATUS will return 00426.
```

Figure 209. ALLOC Operation

ANDxx (And)

Free-Form Syntax	(not allowed - use the AND operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
ANDxx	Comparand	Comparand				

The ANDxx operation must immediately follow one of the following operations:
- ANDxx
- DOUxx
- DOWxx
- IFxx
- ORxx
- WHENxx

With ANDxx, you can specify a complex condition for the DOUxx, DOWxx, IFxx, and WHENxx operations. The ANDxx operation has higher precedence than the ORxx operation. See Figure 211 on page 507 for an example.

Factor 1 and factor 2 must contain a literal, a named constant, a figurative constant, a table name, an array element, a data structure name, or a field name. Factor 1 and factor 2 must be of the same type. For example, a character field cannot be compared with a numeric. The comparison of factor 1 and factor 2 follows the same rules as those given for the compare operations.

"Compare Operations" on page 357 and "Structured Programming Operations" on page 376 describes the rules for specifying the ANDxx operation.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* If ACODE is equal to A and indicator 50 is on, the MOVE
C* and WRITE operations are processed.
C     ACODE         IFEQ      'A'
C     *IN50         ANDEQ     *ON
C                   MOVE      'A'           ACREC
C                   WRITE     RCRSN
C* If the previous conditions were not met but ACODE is equal
C* to A, indicator 50 is off, and ACREC is equal to D, the
C* following MOVE operation is processed.
C                   ELSE
C     ACODE         IFEQ      'A'
C     *IN50         ANDEQ     *OFF
C     ACREC         ANDEQ     'D'
C                   MOVE      'A'           ACREC
C                   ENDIF
C                   ENDIF
```

Figure 210. ANDxx Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* In the following example, indicator 25 will be set on only if the
C* first two conditions are true or the third condition is true.
C*
C* As an expression, this would be written:
C* EVAL *IN25 = ((FIELDA > FIELDB) AND (FIELDA >= FIELDC)) OR (FIELDA < FIELDD)
C*
C*
C     FIELDA        IFGT      FIELDB
C     FIELDA        ANDGE     FIELDC
C     FIELDA        ORLT      FIELDD
C                   SETON                                        25
C                   ELSE
C                   SETOFF                                       25
C                   ENDIF
```

Figure 211. Example of AND/OR Precedence

BEGACT (Begin Action Subroutine)

Free-Form Syntax	BEGACT *action-subroutine-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
BEGACT	Part name	Event name	Window name			

The BEGACT operation defines the start of an action subroutine. When an event for a part occurs, the action subroutine is called.

In the GUI Designer, you can link an event for a part to an existing action subroutine, or create a new action subroutine for the event when one is not defined. In the latter case, a skeleton action subroutine is inserted in the source code, with a name derived from the associated window, part, and event names.

The skeleton action subroutine will be added to the program source code in a free-form style when the location where the subroutine is added falls within a free-form section: ie. when a '/END-FREE' directive follows the end of the Calculation specifications, but before any Output or Procedure or compile-time data specifications.

Action Subroutine Names in Traditional Syntax

When the GUI Designer inserts a new action subroutine in a traditional syntax calculation section, it specifies the part name in Factor1, event name in Factor 2, and the window name (containing the part) in the Result field of the BEGACT operation. The action subroutine name is derived from these values, and linked to a part event.

The action subroutine name is built using factor 1, factor 2, and the result field. Each entry is separated by a plus (+) character. The following table shows examples of links created using the GUI Designer:

Table 53. Single-link and Multiple-link Action Subroutines

Window	Part	Event	Action subroutine
INVENTORY	PSB0001	PRESS	PSB0001+PRESS+INVENTORY
INVENTORY	PSB0004	PRESS	SETCOLORS
INVENTORY	PSB0005	PRESS	PSB0005+PRESS+INVENTORY
ADDPART	PSB0008	PRESS	SETCOLORS
INVENTORY	PSB0002	PRESS	PSB0002++INVENTORY
INVENTORY	PSB0002	MOUSEMOVE	PSB0002++INVENTORY
ADDPART	PSB0009	MOUSEMOVE	PSB0009+MOUSEMOVE

The following examples illustrate how an action subroutine name is built, using the information described in Table 53.

Action Subroutine Names using Factor 1 and Factor 2

If factor 1 contains PSB0009, factor 2 contains MOUSEMOVE, and the result field does not contain an entry, the action subroutine name is PSB0009+MOUSEMOVE.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C     PSB0009       BEGACT    MOUSEMOVE
```

Figure 212. Action Subroutine Name - Factor 1 and Factor 2

Action Subroutine Names using Factor 1 and the Result Field

If factor 1 contains PSB0002, factor 2 does not contain an entry, and the result field contains INVENTORY, the action subroutine name is PSB0002++INVENTORY.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C     PSB0002       BEGACT                  INVENTORY
```

Figure 213. Action Subroutine Name - Factor 1 and Result field

Action Subroutine Names using Factor 1, Factor 2 and Result Field

If factor 1 contains PSB0001, factor 2 contains PRESS, and the result field contains INVENTORY, the action subroutine name is PSB0001+PRESS+INVENTORY.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C     PSB0001       BEGACT    PRESS         INVENTORY
C     PSB0005       BEGACT    PRESS         INVENTORY
```

Figure 214. Action Subroutine Name - Factor 1, Factor 2, and Result Field

Action Subroutine Names using Factor 1

If factor 1 contains SETCOLORS, and both factor 2 and the result field do not contain entries, the action subroutine name is SETCOLORS. This name is used to retrieve the information about the window(s), part(s), and event(s) linked to the action subroutine SETCOLORS.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
C     SETCOLORS     BEGACT
```

Figure 215. Action Subroutine Name - Factor 1

Action Subroutine Names in Free-Form Syntax

The name for an action subroutine inserted in a free-form section is one value generated by joining the window, part, and event names together, separated by underscores.

Sample Action Subroutine in Free-Form Syntax:

```
/free

 //*****************************************************************
 //
```

```
 // Window . . :  DATEDIALOG
 // Part . . . :  PSBOK
 // Event  . . :  CREATE
 //
 // Description:
 //
 //*****************************************************************

 BEGACT DATEDIALOG_PSBOK_CREATE;

 ENDACT;
/end-free
*****END OF SOURCE***
```

Single-Link and Multiple-Link Action Subroutines

Action subroutines that are linked to only one window/part/event combination are called single-link action subroutines.

Action subroutines that are linked to more than one window/part/event combination are called multiple-link action subroutines.

Note: All user subroutines are considered to be multiple-link action subroutines. At runtime, the default window or event for user subroutines is the default window or event of the action subroutine which calls the user subroutine, either directly or through other user subroutines.

Table 53 on page 508 illustrates single-link and multiple-link action subroutines. For example, items 1, 3, and 7 are single-link action subroutines. Items 2 and 4, and items 5 and 6 are multiple-link action subroutines.

Use the following guidelines when working with action subroutines:

- Duplicate action subroutine names are not allowed. Your program cannot contain duplicate action subroutine names. If factor 1 is the only entry for the BEGACT operation, it cannot be the same as any field name, user subroutine name, or the name of any other construct in your program.
- Action subroutines with no events associated are never executed. This can occur if you remove the action link using the GUI Designer.

You use the GUI Designer to create action subroutines and to link each action subroutine to at least one window/part/event combination. When an action subroutine is compiled, the compiler refers to the links that you created using the GUI Designer. You can either use the action subroutine names created by the GUI Designer or you can replace these with your own names. For more information on using the GUI Designer to create and link action subroutines, see *Getting Started with WebSphere Development Studio Client for iSeries*.

BEGSR (Begin User Subroutine)

Free-Form Syntax	BEGSR *subroutine-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
BEGSR	subroutine-name					

The BEGSR operation identifies the beginning of a user subroutine.

Subroutine-name must specify a unique symbolic name or one of the following keywords: *TERMSR, *PSSR or *INZSR. If you specify a name, you must specify the same name in the EXSR operation referring to the subroutine or in the result field of the CASxx operation referring to the subroutine.

If you specify a keyword, only one subroutine can be defined by these keywords:
- *TERMSR specifies a subroutine to be run during normal termination.
- *PSSR specifies that this is a program exception/error subroutine to handle program-detected exception/errors.
- *INZSR specifies a subroutine to be run during initialization.

"EXSR (Invoke User Subroutine)" on page 577 describes how to invoke subroutines.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C     *TERMSR       BEGSR
C                     .
C                     .
C                     .
C                   ENDSR
```

Figure 216. Begin User Subroutine Operation

Note: When referencing parts in a subroutine, consider the following: All user subroutines are considered to be multiple-link action subroutines. At runtime, the default window or event for user subroutines is the default window or event of the action subroutine which calls the user subroutine, either directly or through other user subroutines.

BITOFF (Set Bits Off)

Free-Form Syntax	(not allowed - use the%BITAND and %BITNOT built-in functions. See Figure 139 on page 414.)

Code	Factor 1	Factor 2	Result Field	Indicators		
BITOFF		Bit numbers	Character field			

The BITOFF operation causes bits identified in factor 2 to be set off (set to 0) in the result field. Bits not identified in factor 2 remain unchanged. When BITOFF is used to format a character, you should use both BITON and BITOFF: BITON specifies the bits to be set on (set to 1), and BITOFF specifies the bits to be set off (set to 0). Unless you explicitly set on or set off all the bits in the character, you might not get the character you want.

Factor 2 can contain:

- *Bit numbers 0-7:* From 1 to 8 bits can be set off per operation. They are identified by the numbers 0 through 7. (0 is the leftmost bit.) Enclose the bit numbers in apostrophes. For example, to set off bits 0, 2, and 5, enter '025' in factor 2.
- *Field name:* Specify the name of a one-position character field, table element, or array element in factor 2. The bits that are on in the field, table element, or array element are set off in the result field; bits that are off do not affect the result.
- *Hexadecimal literal or named constant:* Specify a 1 byte hexadecimal literal or hexadecimal named constant. Bits that are on in factor 2 are set off in the result field; bits that are off are not affected.
- *Named constant:* Specify a character named constant up to eight positions long containing the bit numbers to be set off.

In the result field, specify a one-position character field. It can be an array element if each element in the array is a one-position character field.

For more information, see "Bit Operations" on page 352.

See Figure 217 on page 514 for an example of the BITOFF and BITON operations.

If you want to assign a particular bit pattern to a character field, use the MOVE operation with a hexadecimal literal in factor 2.

BITON (Set Bits On)

Free-Form Syntax	(not allowed - use the %BITOR built-in function. See Figure 139 on page 414.)

Code	Factor 1	Factor 2	Result Field	Indicators		
BITON		Bit numbers	Character field			

The BITON operation causes bits identified in factor 2 to be set on (set to 1) in the result field. Bits not identified in factor 2 remain unchanged. When BITON is used to format a character, you should use both BITON and BITOFF: BITON to specify the bits to be set on (set to 1) and BITOFF to specify the bits to be set off (set to 0). Unless you explicitly set on or off all the bits in the character, you might not get the character you want.

Factor 2 can contain:

- *Bit numbers 0-7:* From 1 to 8 bits can be set on per operation. They are identified by the numbers 0 through 7. (0 is the leftmost bit.) Enclose the bit numbers in apostrophes. For example, to set bits 0, 2, and 5 on, enter '025' in factor 2.
- *Field name:* You can specify the name of a one-position character field, table element, or array element in factor 2. The bits that are on in the field, table element, or array element are set on in the result field; bits that are off are not affected.
- *Hexadecimal literal or named constant:* You can specify a 1-byte hexadecimal literal. Bits that are on in factor 2 are set on in the result field; bits that are off do not affect the result.
- *Named constant:* You can specify a character named constant up to eight positions long containing the bit numbers to be set on.

In the result field, specify a one-position character field. It can be an array element if each element in the array is a one-position character field.

For more information, see "Bit Operations" on page 352.

See Figure 217 on page 514 for an example of the BITOFF and BITON operations.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D BITNC           C                   '01234567'
D HEXNC           C                   X'0F'
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The bit settings are:
C*  Before the operations:       After the operations:

C*          FieldA = 00000000    FieldA = 10001111
C*          FieldB = 00000000    FieldB = 00010000
C*          FieldC = 11111111    FieldC = 11111111
C*          FieldD = 11000000    FieldD = 11010000
C*          FieldE = 11000000    FieldE = 11000001
C*          FieldG = 11111111    FieldG = 01111111
C*          FieldH = 00000000    FieldH = 00001110
C*          FieldI = 11001010    FieldI = 00001111
C*
C                   BITON     '04567'       FieldA
C                   BITON     '3'           FieldB
C                   BITON     '3'           FieldC
C                   BITON     '3'           FieldD
C                   BITON     '01'          FieldH
C                   BITOFF    '0'           FieldG
C                   BITOFF    BITNC         FieldI
C                   BITON     HEXNC         FieldI
```

Figure 217. BITON and BITOFF Operations

If you want to assign a particular bit pattern to a character field, use the MOVE operation with a hexadecimal literal in factor 2.

CABxx (Compare and Branch)

Free-Form Syntax	(not allowed - use other operation codes, such as LEAVE, ITER, and RETURN)

Code	Factor 1	Factor 2	Result Field	Indicators		
CABxx	Comparand	Comparand	Label	HI	LO	EQ

The CABxx operation compares factor 1 with factor 2. If the condition specified by xx is true, the program branches to the TAG or ENDSR operation associated with the label specified in the result field. Otherwise, the program continues with the next operation in the sequence. If the result field is not specified, the resulting indicators are set accordingly, and the program continues with the next operation in the sequence.

"Compare Operations" on page 357 describes the different values for xx.

Factor 1 and factor 2 must contain a literal, a named constant, a figurative constant, a table name, an array element, a data structure name, or a field name. Factor 1 and factor 2 must be of the same type.

A CABxx operation in the main procedure can specify a branch to a previous or a succeeding specification line. A CABxx operation in a subprocedure can specify a branch:

- From a line in the body of the subprocedure to another line in the body of the subprocedure
- From a line in a subroutine to another line in the same subroutine
- From a line in a subroutine to a line in the body of the subprocedure

The CABxx operation cannot specify a branch from outside a subroutine to a TAG or ENDSR operation within that subroutine. The label specified in the result field must be associated with a unique TAG operation and must be a unique symbolic name.

Resulting indicators are optional. When specified, they are set to reflect the results of the compare operation. For example:

- HI is set when factor 1 is greater than factor 2
- LO is set when factor 1 is less than factor 2
- EQ is set when factor 1 and factor 2 are equal.

For more information, see "Branching Operations" on page 352.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*         The field values are:
C*         FieldA = 100.00
C*         FieldB = 105.00
C*         FieldC = ABC
C*         FieldD = ABCDE
C*
C*          Branch to TAGX.
C     FieldA        CABLT     FieldB        TAGX
C*
C*          Branch to TAGX.
C     FieldA        CABLE     FieldB        TAGX
C*
C*          Branch to TAGX; indicator 16 is off.
C     FieldA        CABLE     FieldB        TAGX                    16
C*
C*          Branch to TAGX; indicator 17 is off, indicator 18 is on.
C     FieldA        CAB       FieldB        TAGX                  1718
C*
C*          Branch to TAGX; indicator 19 is on.
C     FieldA        CAB       FieldA        TAGX                     19
C*
C*          No branch occurs.
C     FieldA        CABEQ     FieldB        TAGX
C*
C*          No branch occurs; indicator 20 is on.
C     FieldA        CABEQ     FieldB        TAGX                   20
C*
C*          No branch occurs; indicator 21 is off.
C     FieldC        CABEQ     FieldD        TAGX                  21
C                   :
C     TAGX          TAG
```

Figure 218. CABxx Operations

CALL (Call an AS/400 Program)

Free-Form Syntax	(not allowed - use the CALLP operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
CALL (E)		Program name	Plist name	_	ER	_

The CALL operation passes control to an AS/400 program represented by the program name specified in factor 2.

Factor 2 must be the name of a definition specification which defines the name of the program to be called. The program name can either be the OS/400 name (optionally library qualified) or an override name you defined using the Define server information menu item. For more information on using the GUI Designer to define server information, see *Programming with VisualAge RPG* and the online help.

If the result field is specified, it must contain the name of a PLIST to communicate values between the calling program and the called program. The result field can be blank if the called program does not access parameters, or if the PARM statements directly follow the CALL operation.

The parameters associated with a CALL to an OS/400 program have the following restrictions:

- Parameters cannot contain a pointer. If a parameter does contain a pointer, the compiler generates an error message at compile time.
- A data structure cannot have overlapping non-character fields. Any overlapping fields must both be character.
- Passing the value *HIVAL (X'FF') as a character or graphic parameter may cause unpredictable results.
- Programs with remote calls that pass in a character field which cannot be converted to EBCDIC, cause translation to stop. Typically, this can occur when a numeric field overlays a character field.
- You can specify a maximum of 25 parameters.
- The total number of bytes allocated for the parameters cannot exceed 32K.

If a resulting indicator is specified in positions 73 and 74, it is set on when an error occurs during the CALL operation.

To handle CALL exceptions (program status codes 202, 211, or 231), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Functions+++++++++++++++++++++++++
D                                   Functions-cont++++++++++++++++++++
D
D*Named Constant
D Remote1         C                   CONST('PROG1')
D                                     LINKAGE(*SERVER) NOWAIT
D*
D*Stand alone field
D Remote2         S             13A   INZ('MYLIB/REMPROG')
D                                     LINKAGE(*SERVER)
D parm1           S              8P 2
D parm2           DS
D name                     1     20A
D   first                  1      8A
D   last                   9     20A
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C* CALL to a remote program
C*
C                   CALL      Remote1                               90
C                   PARM                    parm1
C                   PARM                    parm2
C*
C* Remote call
C*
C                   CALL      Remote2       PLIST1                  90
C*
C*
C*
C     PLIST1        PLIST
C                   PARM                    Fld1            10 2
C                   PARM                    Charfld         50
```

Figure 219. CALL Operation

Calling an OS/400 Program that Uses a Workstation File

Do the following to use a VisualAge RPG program that calls an OS/400 program that uses a workstation file:

- Specify the NOWAIT keyword in the Definition specification
- When you create the OS/400 workstation file on the server, specify the following for the CRTDSPF command:
  - Display Device value: the name of the session where the diplay file is to be displayed.
  - Maximum Number of Devices: any value greater than 1.
- In the remote AS/400 program, **do not use** the ACQ operation to acquire a display device. Doing this will cause a conflict that will result in an error.

Note: When using this method, you can pass parameters to the remote program. However, no parameters can be returned from the remote program.

Calling Host Programs that Use Display Files

When the VARPG compiler calls an OS/400 host program that uses display files, determination of a valid session device that can be used is necessary. To determine a valid session device that can be used by the host program, you can use a CL program on the host to locate a valid session.

The following example illustrates such a CL program. It assumes that you are using the SNA protocol with 5250 or Graphical Access emulation running on Client Access.

```
PGM  PARM(&SESS)
/*-------------------------------------------------------*/
/*                                                       */
/* DECLARE WORKING VARIABLES                             */
/*                                                       */
/*-------------------------------------------------------*/
 DCL  VAR(&JOBN) TYPE(*CHAR) LEN(10)
 DCL  VAR(&SESS) TYPE(*CHAR) LEN(10)
 DCL  VAR(&SUB)  TYPE(*CHAR) LEN(2)
 DCL  VAR(&STS)  TYPE(*DEC) LEN(5 0)
 DCL  &ITLEN     TYPE(*DEC)  VALUE(2)
 DCL  &ITPTR     TYPE(*DEC)  LEN(5 0)
 DCL  VAR(&SUBFIX) TYPE(*CHAR) LEN(40) +
      VALUE('A B C D E F G H I J G0G1G2G3G4G5G6G7G8G9')

 RTVJOBA    JOB(&JOBN)
/*-------------------------------------------------------*/
/* LOOP THROUGH THE POSSIBLE DEVICE NAME AND             */
/*   CHECK IF THERE IS ONE WITH SIGNON DISPLAY ON.       */
/*-------------------------------------------------------*/
 CHGVAR    &ITPTR 1

LOOP1:
      IF (&ITPTR *GT 40) THEN(DO)
         CHGVAR &SESS VALUE('INVALID   ')
         GOTO END
      ENDDO

      CHGVAR     VAR(&SUB)  VALUE(%SST(&SUBFIX &ITPTR &ITLEN))
      CHGVAR     VAR(&SESS) VALUE(&JOBN *TCAT &SUB)
      RTVCFGSTS  CFGD(&SESS) CFGTYPE(*DEV) STSCDE(&STS)
      MONMSG     MSGID(CPF9801)
      IF  (&STS = 50) THEN(GOTO END)
      CHGVAR    &ITPTR (&ITPTR + &ITLEN)
      GOTO      LOOP1
END:  ENDPGM
```

Calling CL Commands

If the VisualAge RPG program calls CL commands:,

- Specify a CALL to QCMDDDM if the CL command issues commands for OS/400 files
- Specify a CALL to QCMDEXC if the CL command issues commands to OS/400 programs and/or data areas.

CALLB (Call a Function)

Free-Form Syntax	(not allowed - use the CALLP operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
CALLB (D E)		Procedure name or procedure pointer	Plist name	_	ER	_

Use CALLB to call a Windows function. Functions are exported names from dynamic link libraries (DLLs) which are linked to the VisualAge RPG application when the application is compiled. For information on how to compile an application that calls a Windows C function, see *Getting Started with WebSphere Development Studio Client for iSeries*.

Factor 2 must contain a procedure name or a procedure pointer containing the address of the function to be called.

The procedure name is case sensitive. This means that the name entered in factor 2 must match the case of the function being called. The procedure name must be 255 characters or less. If the name is longer than 255, it is truncated to 255.

If factor 2 contains a procedure pointer, the *ROUTINE in the PSDS is cleared and filled with blanks. If factor 2 contains a literal or named constant, *ROUTINE in the PSDS contains the first eight characters of the procedure name.

If the result field is specified, it must contain a PLIST name.

If a resulting indicator is specified in positions 73 and 74, it is set on when an error occurs during the CALLB operation.

To handle CALLB exceptions (program status codes 202, 211, or 231), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

The linkage convention, __cdecl,must be used in the called function.

Note: The VisualAge RPG compiler uses this linkage convention for VARPG subprocedures.

See *Programming with VisualAge RPG* for examples of how to use the CALLB operation.

CALLP (Call a Prototyped Procedure or Program)

Free-Form Syntax	{CALLP{(EMR)}} *name*({*parm1*{:*parm2*...}})

Code	Factor 1	Factor 2
CALLP (M/R)		name{ (Parm1 {:Parm2...}) }

The CALLP operation is used to call prototyped procedures, local workstation programs (.EXE, .BAT, or .COM's), or remote programs on an iSeries server. This is the recommended way of calling programs.

UCS-2 parameters are not allowed.

Unlike the other call operations, CALLP uses a free-form syntax. You use the *name* operand to specify the name of the prototype of the called program or procedure, as well as any parameters to be passed. (This is similar to calling a built-in function.) A maximum of 255 parameters are allowed for a program call, and a maximum of 399 for a procedure call.

On a free-form calculation specification, the operation code name may be omitted if no extenders are needed.

A prototype for the program or procedure being called must be included in the definition specifications preceding the CALLP. The compiler uses the prototype name to obtain an external name, if required, for the call.

If CALLP is used to call a procedure which returns a value, that value will not be available to the caller. If the value is required, call the prototyped procedure from within an expression.

To perform a dynamic external call to a local workstation program, specify keyword CLTPGM on the prototype. With this type of call, there can be no return value and parameters must be passed by value.

To perform a remote call to an iSeries program, specify EXTPGM(*program-name*) and LINKAGE(*SERVER) on the prototype.

For information on how to define a local program and for the rules for passing parameters, see Chapter 18, "Definition Specifications," on page 255. For information on procedures, subprocedures, and prototyping, see Chapter 6, "Subprocedures and Prototypes," on page 63. See *Programming with VisualAge RPG* for information on calling programs and using multiple procedures.

For more information on call operations, see "Call Operations" on page 353. For more information on defining prototypes, see "Prototypes and Parameters" on page 71. For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 390.

Note: Programs that are called using CALLP complete execution before any statements after CALLP are executed.

In the following example, the parameter fld1 is passed to program pgm1.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
D                                    Functions-cont++++++++++++++++++++
D pgm1            PR                 CLTPGM('testprog')
D fld1                            20A VALUE
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C*
C                   CALLP     pgm1(fld1)                            90
```

Figure 220. CALLP Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
 * Remote program call requires LINKAGE(*SERVER) and EXTPGM()
d target1         pr                 linkage(*server)
d                                    extpgm('TARGET1')
d                                 20
d                                  9s 2

 /free
       target1( p1: p2);
```

Figure 221. Remote prototyped call to server program

```
 * The prototype for the procedure has an array parameter.
D proc            pr
D   parm                        10a   dim(5)

* An array to pass to the procedure
D array           s             10a   dim(5)

* Call the procedure, passing the array
C                   callp     proc (array)
```

Figure 222. Passing an array parameter using CALLP

CASxx (Conditionally Invoke Subroutine)

Free-Form Syntax	(not allowed - use the IF and EXSR operation codes)

Code	Factor 1	Factor 2	Result Field	Indicators		
CASxx	Comparand	Comparand	Subroutine name	HI	LO	EQ

The CASxx operation is used to conditionally select a subroutine for processing. The selection is based on the relationship between factor 1 and factor 2, as specified by xx. If the relationship denoted by xx exists between factor 1 and factor 2, the subroutine specified in the result field is processed. If the relationship denoted by xx does not exist, the program continues with the next CASxx operation in the CAS group. For a list of xx values, see "Compare Operations" on page 357.

A CAS group can contain only CASxx operations. An ENDCS operation must follow the last CASxx operation. After the subroutine is processed, the program continues with the next operation following the ENDCS operation, unless the subroutine passes control to a different operation.

If factor 1 and factor 2 are specified, they can contain a literal, a named constant, a figurative constant, a field name, a table name, an array element, a data structure name, or blanks. Both factor 1 and factor 2 must be of the same data type. Blanks are valid only if xx is blank and no resulting indicators are specified.

The result field must contain the name of a user subroutine or one of the following the keywords: *TERMSR, *PSSR or *INZSR:
- *TERMSR specifies a subroutine to be run during normal termination.
- *PSSR specifies that this is a program exception/error subroutine to handle program-detected exception/errors.
- *INZSR specifies a subroutine to be run during initialization.

Conditioning indicators can be specified for the CASxx operation, however, conditioning indicators cannot be specified on the ENDCS operation for a CAS group.

In a CASbb operation, factor 1 and factor 2 are required only if resulting indicators are specified in positions 71 through 76. The CASbb operation with no resulting indicators specified in positions 71 through 76 is functionally identical to an EXSR operation, because it causes the unconditional running of the subroutine named in the result field of the CASbb operation. Any CASxx operations that follow an unconditional CASbb operation in the same CAS group are never tested. Therefore, the normal placement of the unconditional CASbb operation is after all other CASxx operations in the CAS group.

If resulting indicators are specified, they are set on as follows:
- High: (71-72) Factor 1 is greater than factor 2.
- Low: (73-74) Factor 1 is less than factor 2.
- Equal: (75-76) Factor 1 equals factor 2.

See "Compare Operations" on page 357 or "Subroutine Operations" on page 378 for further rules for the CASxx operation.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The CASGE operation compares FieldA with FieldB.  If FieldA is
C*  greater than or equal to FieldB, Subr01 is processed and the
C*  program continues with the operation after the ENDCS operation.
C*
C     FieldA        CASGE     FieldB        Subr01
C*
C*  If FieldA is not greater than or equal to FieldB, the program
C*  next compares FieldA with FieldC.  If FieldA is equal to FieldC,
C*  SUBR02 is processed and the program continues with the operation
C*  after the ENDCS operation.
C*
C     FieldA        CASEQ     FieldC        Subr02
C*
C*  If FieldA is not equal to FieldC, the CAS operation causes Subr03
C*  to be processed before the program continues with the operation
C*  after the ENDCS operation.
C*  The CAS statement is used to provide a subroutine if none of
C*  the previous CASxx operations have been met.
C*
C                   CAS                   Subr03
C*
C*  The ENDCS operation denotes the end of the CAS group.
C*
C                   ENDCS
```

Figure 223. CASxx Operations

CAT (Concatenate Two Strings)

Free-Form Syntax	(not allowed - use the + operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
CAT (P)	Source string 1	Source string 2: number of blanks	Target string			

The CAT operation concatenates the string specified in factor 2 to the end of the string specified in factor 1 and places it in the result field. The source and target strings must all be of the same type, either all character, all graphic, or all UCS-2.

If factor 1 is specified, it must contain a string which can be a field name, array element, named constant, data structure name, table name, or literal. If no factor 1 is specified, factor 2 is concatenated to the end of the result field string.

Note: In the following description of the CAT operation, references to factor 1 apply to the result field if factor 1 is not specified.

Factor 2 must contain a string, and may contain the number of blanks to be inserted between the concatenated strings. Its format is the string, followed by a colon, followed by the number of blanks. The blanks are in the format of the data. For example, for character data a blank is x'20', while for UCS-2 data a blank is x'0020'. If graphic strings are being concatenated, the blanks are double-byte blanks. The string portion can contain a field name, array element, named constant, data structure name, table name, literal, or data structure subfield name. The number of blanks must be numeric with zero decimal positions, and can contain a named constant, array element, literal, table name, or field name.

If a colon is specified, the number of blanks must be specified. If no colon is specified, concatenation occurs with the trailing blanks, if any, in factor 1, or the result field if factor 1 is not specified.

If the number of blanks (N) is specified, factor 1 is copied to the result field left-justified. If factor 1 is not specified the result field string is used. N blanks are then added following the last nonblank character. Factor 2 is then appended to this result. Leading blanks in factor 2 are not counted when N blanks are added to the result; they are just considered to be part of factor 2. If the number of blanks is not specified, the trailing and leading blanks of factor 1 and factor 2 are included in the result.

The result field must be a string and can contain a field name, array element, data structure name, or table name. Its length should be the length of factor 1 and factor 2 combined plus any intervening blanks; if it is not, truncation occurs from the right.

A P operation extender indicates that the result field should be padded on the right with blanks after the concatenation occurs if the result field is longer than the result of the operation. If padding is not specified, only the leftmost part of the field is affected.

At run time, if the number of blanks is fewer than zero, the compiler defaults the number of blanks to zero.

Figurative constants cannot be used in the factor 1, factor 2, or result fields. No overlapping is allowed in a data structure for factor 1 and the result field, or for factor 2 and the result field.

"String Operations" on page 375 describes the general rules for specifying string operations.

For more information, see "String Operations" on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* CAT concatenates LAST to NAME and inserts one blank as specified
C* in factor 2.  TEMP contains 'Mr.bSmith'.
C                   MOVE      'Mr.   '      NAME              6
C                   MOVE      'Smith '      LAST              6
C     NAME          CAT       LAST:1        TEMP              9
C*
C* CAT concatenates 'OS' to STRING and places 'OSXX' in TEMP.
C                   MOVE     'XX'         STRING             2
C     'OS'          CAT      STRING       TEMP               4
C*
C* The following example is the same as the previous example except
C* that TEMP is defined as a 10 byte field.  P operation extender
C* specifies that blanks will be used in the rightmost positions
C* of the result field that the concatenation result, 'OSXX',
C* does not fill.  As a result, TEMP contains 'OSXXbbbbbb'
C* after concatenation.
C                   MOVE      *ALL'*'       TEMP              10
C                   MOVE      'XX'          STRING             2
C     'OS'          CAT(P)    STRING        TEMP
C*
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following example shows leading blanks in factor 2.  After
C* the CAT the RESULT contains 'MR.bSMITH'.
C*
C                   MOVE      'MR.'         NAME              3
C                   MOVE      ' SMITH'      FIRST             6
C     NAME          CAT       FIRST         RESULT            9
C*
C*  The following example shows the use of CAT without factor 1.
C*  FLD2 is a 9 character string.  Prior to the concatenation, it
C*  contains 'ABCbbbbbb.' FLD1 contains 'XYZ'.  After the
C*  concatenation FLD2 contains 'ABCbbXYZb'.
C*
C                   MOVEL(P) 'ABC'          FLD2              9
C                   MOVE     'XYZ'          FLD1              3
C                   CAT      FLD1:2         FLD2
```

Figure 224. CAT Operations

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *
 * The following example shows the use of graphic strings
 *
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
 *       Value of Graffld  is 'AACCBBGG'.
 *       Value of Graffld2 after CAT 'aa    AACCBBGG      '
 *       Value of Graffld3 after CAT 'AABBCCDDEEFFGGHHAACC'
 *
D Graffld                         4G   INZ(G'AACCBBGG')
D Graffld2                       10G   INZ
D Graffld3                       10G   INZ(G'AABBCCDDEEFFGGHH')
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
 * The value 2 represents 2 graphic blanks as separators
C     G'aa'         cat       Graffld:2     Graffld2
C                   cat       Graffld       Graffld3
```

Figure 225. CAT Operation with Graphic Data

CHAIN (Random Retrieval from a File)

Free-Form Syntax	CHAIN{(ENHMR)} *search-arg name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
CHAIN (E N)	search-arg	name (file or record format)	data-structure	NR	ER	_

The CHAIN operation retrieves a record from a full procedural file (F in position 18 of the file description specifications) or a subfile, sets a record identifying indicator on (if specified on the input specifications), and places the data from the record into the input fields.

Retrieving Data from a File or Record Format

The file must be specified on the file description specifications. It can be a remote server file or a local file.

The search argument, *search-arg*, must be the key or relative record number used to retrieve the record. If access is by key, *search-arg* can be a a single key in the form of a field name, a named constant, a figurative constant, or a literal.

The search argument, *search-arg*, must be the key, relative record number, or KLIST name used to retrieve the record:

- If access is by key, the *name* operand must be a remote OS/400 file. The search argument can be a single key in the form of a field name, a named constant, a figurative constant, or a literal.

  If the file is an externally-described file, *search-arg* can also be a composite key in the form of a KLIST name, a list of values, or %KDS. Graphic and UCS-2 key fields must have the same CCSID as the key in the file. For an example of %KDS, see the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451.
- If access is by relative record number, the search argument must specify an integer literal or a numeric field with zero decimal positions.

The *name* operand specifies the file or record format name that is to be read:

- If *name* is a file name, the first record that matches the search argument is retrieved.
- If *name* is an OS/400 file name and *MBR ALL is specified, only the current open file member is processed.
- If *name* is a local disk file, it must be program described.
- If *name* is a record format name, the file can be externally described.
- If *name* is a record format name and access is by key, the first record of the specified record type whose key matches the search argument is retrieved.

Note: Record locking is supported for OS/400 remote files. Record locking is not supported for local files.

If the *data-structure* operand is specified, the record is read directly into the data structure. If *name* refers to a program-described file (identified by an F in position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or

LIKEREC(...:*INPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

If the file specified in *name* is an OS/400 input DISK file, no operation extender is allowed. All records are read without locks.

If the file specified in *name* is an OS/400 UPDATE file, and if the operation extender N is not specified the CHAIN operation locks a record. The record remains locked until:
- The record is updated
- The record is deleted
- Another record is read from the file for input or update
- A SETLL or SETGT is performed on the file
- An UNLOCK operation is performed on the file
- An output operation defined by an output specification with no field names is performed on the file.

An output operation that adds a record to a file does not cause a record lock to be released.

You can specify an indicator in positions 71-72 that is set on if no record in the file matches the search argument. This information can also be obtained from the %FOUND built-in function, which returns '0' if no record is found, and '1' if a record is found.

To handle CHAIN exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

Positions 75 and 76 must be blank.

When the CHAIN operation is successful, the file is positioned so that a subsequent read operation retrieves the record logically following or preceding the retrieved record. When the CHAIN operation does not complete successfully, the fields in the program remain unchanged and the file must be repositioned before a subsequent read operation can be done on the file.

If the file is updated immediately after a successful CHAIN operation, the last record retrieved is updated.

If a record is not found, if an error occurs during the CHAIN operation, or if the last record has already been retrieved (end of file), no data is retrieved and all fields remain unchanged.

For more information, see "File Operations" on page 363.

Note: Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
*
*  The CHAIN operation retrieves the first record from the file,
*  FILEX, that has a key field with the same value as the search
*  argument KEY (factor 1).

/FREE
    CHAIN  KEY  FILEX;

 //  If a record with a key value equal to the search argument is
 //  not found, %FOUND returns '0' and the EXSR operation is
 //  processed. If a record is found with a key value equal
 //  to the search argument, the program continues with
 //  the calculations after the EXSR operation.

    IF  NOT %FOUND;
       EXSR  Not_Found;
    ENDIF;
/END-FREE
```

Figure 226. CHAIN Operation with a File Name

```
FFilename++IPEASF.....L.....A.Device+.Keywords++++++++++++++++++++++++++
FCUSTFILE   IF   E           K DISK    REMOTE
 /free
        // Specify the search keys directly in a list
        chain ('abc' : 'AB') custrec;
        // Expressions can be used in the list of keys
        chain (%xlate(custname : LO : UP) : companyCode + partCode)
                custrec;
        return;
```

Figure 227. CHAIN Operation Using a List of Key Fields

```
FFilename++IPEASF.....L.....A.Device+.Keywords++++++++++++++++++++++++++
FCUSTFILE   IF   E           K DISK      REMOTE
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D custRecDs       ds                  likerec(custRec)

 /free
        // Read the record directly into the data structure
        chain ('abc' : 'AB') custRec custRecDs;
        // Use the data structure fields
        if (custRecDs.code = *BLANKS);
           custRecDs.code = getCompanyCode (custRecDs);
           update custRec custRecDs;
        endif;
```

Figure 228. CHAIN Operation Using a Data Structure with an Externally-Described File

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The CHAIN operation uses the value contained in the search
C*  argument KEY to retrieve a record of the record type REC1 from
C*  an externally described file. If no record is found of the
C*  specified type that has a key field equal to the search
C*  argument, indicator 72 is set on.  A complex key with a KLIST is
C*  used to retrieve records from files that have a composite key.
C*  If a record of the specified type is found that has a key field
C*  equal to the search argument, indicator 72 is set off and therefore
C*  the UPDATE operation is processed.
C*
C     KEY           CHAIN     REC1                              72
C     KEY           KLIST
C                   KFLD                Field1
C                   KFLD                Field2
C                   IF        NOT *IN72
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The UPDATE operation modifies all the fields in the REC1 record.
C*
C                   UPDATE    REC1
C                   ENDIF
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The following example shows a CHAIN with no lock.
C*
C                   MOVE 3                Rec_No
C     Rec_No        CHAIN (N) INPUT                             99
```

Figure 229. CHAIN Operation with a Record Format Name and with No Lock

Retrieving a Record from a Subfile Part

If *name* is a subfile part, the search argument must be an index. The CHAIN operation reads a record from a subfile using the index.

Before a record in a subfile part can be updated or deleted, the subfile must be positioned to the record. *START and *END cannot be used with a subfile part. The field values from the subfile part are assigned to the corresponding program values for the subfile fields. These values can be modified by the program.

CHECK (Check Characters)

Free-Form Syntax	(not allowed - use the %CHECK built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
CHECK (E)	Comparator string	Base string:start	Left-position	_	ER	FD

The CHECK operation verifies that each character in the base string (factor 2) is among the characters indicated in the comparator string (factor 1). The base string and comparator string must be of the same type, either both character, both graphic, or both UCS-2. (Graphic and UCS-2 types must have the same CCSID value.) Verifying begins at the leftmost character of factor 2 and continues character by character, from left to right. Each character of the base string is compared with the characters of factor 1. If a match for a character in factor 2 exists in factor 1, the next base string character is verified. If a match is not found, an integer value is placed in the result field to indicate the position of the incorrect character.

The operation stops checking when it finds the first incorrect character or when the end of the base string is encountered. If no incorrect characters are found, the result field is set to zero.

Factor 1 must be a string, and can contain a field name, array element, named constant, data structure name, data structure subfield, literal, or table name.

Factor 2 must contain either the base string or the base string, followed by a colon, followed by the start position. The base string must contain a field name, array element, named constant, data-structure name, literal, or table name. The start position must be numeric with no decimal positions, and can be a named constant, array element, field name, literal, or table name. If no start position is specified, a value of 1 is used. If the start position is greater than 1, the value in the result field is relative to the leftmost position in the base string, regardless of the start position.

If a result field is specified, it can be a numeric variable, numeric array element, numeric table name, or numeric array. If the result field is not specified, you must specify the found indicator in position 75-76.

Do not use decimal positions in the result field.

If the result field is an array, the operation continues checking after the first incorrect character is found for as many occurrences as there are elements in the array. If there are more array elements than incorrect characters, all of the remaining elements are set to zeros. If graphic or UCS-2 data is used, the result field will contain graphic character positions (that is, position 3, the 3rd graphic character, will be character position 5).

To handle CHECK exceptions (program status code 100), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

You can specify an indicator in positions 75-76 that is set on if any incorrect characters are found. This information can also be obtained from the %FOUND built-in function, which returns '1' if any incorrect characters are found.

Figurative constants cannot be used in the factor 1, factor 2, or result fields. No overlapping is allowed in a data structure for factor 1 and the result field or for factor 2 and the result field.

"String Operations" on page 375 describes the general rules for specifying string operations.

For more information, see "String Operations" on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D* After the following example, N=6 and the found indicator 90
D* is on.  Because the start position is 2, the first nonnumeric
D* character found is the '.'.
D*
D
D Digits          C                   '0123456789'
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C
C                   MOVE      '$2000.'      Salary
C     Digits        CHECK     Salary:2      N                       90
C*
C
C                   MOVE      '$2000.'      Salary
C     Digits        CHECK     Salary:2      N
C                   IF        %FOUND
C                   EXSR      NonNumeric
C                   ENDIF
C*
C*
C* Because factor 1 is a blank, CHECK indicates the position
C* of the first nonblank character.  If STRING contains 'bbbthe',
C* NUM will contain the value 4.
C*
C
C           ' '     CHECK     String        Num               2 0
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
.
.
.
```

Figure 230. CHECK Operation (Part 1 of 2)

```
.
.
.
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D* The following example checks that FIELD contains only the letters
D* A to J.  As a result, ARRAY=(136000) after the CHECK operation.
D* Indicator 90 turns on.
D*
D
D Letter          C                   'ABCDEFGHIJ'
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C
C                   MOVE      '1A=BC*'      Field             6
C     Letter        CHECK     Field         Array                    90
C
C*
C* In the following example, because FIELD contains only the
C* letters A to J, ARRAY=(000000).  Indicator 90 turns off.
C*
C
C                   MOVE      'FGFGFG'      Field             6
C     Letter        CHECK     Field         Array                    90
C
C
```

Figure 230. CHECK Operation (Part 2 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D
 * The following example checks a DBCS field for valid graphic
 * characters starting at graphic position 2 in the field.
D
 *       Value of Graffld  is 'DDBBCCDD'.
 *       The value of num after the CHECK is 4, since this is the
 *       first character 'DD' which is not contained in the string.
D
D Graffld                       4G   INZ(G'DDBBCCDD')
D Num                           5 0
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
C
C
C     G'AABBCC'     check     Graffld:2     Num
```

Figure 231. CHECK Operation with Graphic Data

CHECKR (Check Reverse)

Free-Form Syntax	(not allowed - use the %CHECKR built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
CHECKR (E)	Comparator string	Base string:start	Right-position	_	ER	FD

The CHECKR operation verifies that each character in the base string is among the characters indicated in the comparator string. The base string and comparator string must be of the same type, either both character, both graphic, or both UCS-2. (Graphic and UCS-2 types must have the same CCSID value.). Verifying begins at the rightmost character of factor 2 and continues character by character, from right to left. Each character of the base string is compared with the characters of factor 1. If a match for a character in factor 2 exists in factor 1, the next source character is verified. If a match is not found, an integer value is placed in the result field to indicate the position of the incorrect character. Although checking is done from the right, the position placed in the result field will be relative to the left.

Factor 1 must be a string and can contain a field name, array element, named constant, data structure name, data structure subfield, literal, or table name.

Factor 2 must contain either the base string or the base string, followed by a colon, followed by the start position. The base string must contain a field name, array element, named constant, data structure name, data structure subfield name, literal, or table name. The start position must be numeric with no decimal positions, and can be a named constant, array element, field name, literal, or table name. If no start position is specified, the length of the string is used.

If a result field is specified, it can be a numeric variable, numeric array element, numeric table name, or numeric array. If the result field is not specified, you must specify the found indicator in position 75-76. The value in the result field is relative to the leftmost position in the source string, regardless of the start position.

Do not use decimal positions in the result field.

If the result field is an array, the operation continues checking after the first incorrect character is found for as many occurrences as there are elements in the array. If there are more array elements than incorrect characters, all of the remaining elements are set to zeros. If the result field is not an array, the operation stops checking when it finds the first incorrect character or when the end of the base string is encountered. If no incorrect characters are found, the result field is set to zero.

If graphic or UCS-2 data is used, the result field will contain graphic character positions (that is, position 3, the 3rd graphic character, will be character position 5).

To handle CHECKR exceptions (program status code 100), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

You can specify an indicator in positions 75-76 that is set on if any incorrect characters are found. This information can also be obtained from the %FOUND built-in function, which returns '1' if any incorrect characters are found.

Figurative constants cannot be used in the factor 1, factor 2, or result fields. No overlapping is allowed in a data structure for factor 1 and the result field, or for factor 2 and the result field.

For more information, see “String Operations” on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* Because factor 1 is a blank character, CHECKR indicates the
C* position of the first nonblank character.  This use of CHECKR
C* allows you to determine the length of a string.  If STRING
C* contains 'ABCDEF   ', NUM will contain the value 6.
C* If an error occurs, %ERROR is set to return '1' and
C* %STATUS is set to return status code 00100.
C*
C
C     ' '           CHECKR(E) String        Num
C
C                   SELECT
C                   WHEN      %ERROR
C ... an error occurred
C                   WHEN      %FOUND
C ... NUM is less than the full length of the string
C                   ENDIF
C*
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++++
D*
D* After the following example, N=1 and the found indicator 90
D* is on. Because the start position is 5, the operation begins
D* with the rightmost 0 and the first nonnumeric found is the '$'.
D*
D Digits          C                   '0123456789'
D
D*
```

Figure 232. CHECKR Operation (Part 1 of 2)

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C                   MOVE      '$2000.'      Salary           6
C     Digits        CHECKR    Salary:5      N                       90
C
.
.
.
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*
D* The following example checks that FIELD contains only the letters
D* A to J.  As a result, ARRAY=(876310) after the CHECKR operation.
D* Indicator 90 turns on.
D
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D Array           S              1    DIM(6)
D Letter          C                   'ABCDEFGHIJ'
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C                   MOVE      '1A=BC***'    Field             8
C     Letter        CHECKR    Field         Array                   90
C
```

Figure 232. CHECKR Operation (Part 2 of 2)

CLEAR (Clear)

Free-Form Syntax	CLEAR {*NOKEY} {*ALL} *name*

Code	Factor 1	Factor 2	Result Field	Indicators		
CLEAR	*NOKEY	*ALL	name (variable or record format)			
CLEAR			name (window or subfile)			

The CLEAR operation sets the following to their default initialization value depending on field type (numeric, character, graphic, UCS-2, indicator, pointer, or date/time/timestamp):

- Elements in a structure (record formats, data structures, arrays, tables)
- Variables (fields, subfields, array elements, indicators)
- Entry field parts on a window
- Subfiles.

For the default initialization value for a data type, see Chapter 9, "Data Types and Data Formats," on page 103.

Fully qualified names may be specified as the Result-Field operand for CLEAR when coded in free-form calculation specifications. If the structure or variable being cleared is variable-length, its length changes to 0. The CLEAR operation allows you to clear structures on a global basis, as well as element by element, during run time.

*ALL, *NOKEY cannot be specified for windows or subfiles.

See "Initialization Operations" on page 366.

Clearing Variables

You cannot specify *NOKEY.

*ALL is optional. If *ALL is specified and the *name* operand is a multiple occurrence data structure or a table name, all occurrences or table elements are cleared and the occurrence level or table index is set to 1.

The *name* operand specifies the variable to be cleared. The particular entry in the *name* operand determines the clear action as follows:

Single occurrence data structure
All fields are cleared in the order in which they are declared within the structure.

Multiple-occurrence data structure
If *ALL is not specified, all fields in the *current* occurrence are cleared. If *ALL is specified, all fields in *all* occurrences are cleared.

Table name
If *ALL is not specified, the *current* table element is cleared. If *ALL is specified, all table elements are cleared.

Array name
Entire array is cleared

Array element (including indicators)
Only the element specified is cleared.

Clearing Record Formats

If the *name* operand specifies a DISK record format name, *NOKEY can be specified to clear all fields except key fields.

*ALL is optional. If *ALL is specified and *NOKEY is not, all fields in the record format are cleared. If *ALL is not specified, only those fields that are output in that record format are affected. If *NOKEY is specified, then key fields are not cleared, even if *ALL is specified.

The *name* operand is the record format to be cleared. Fields are cleared in the order they are defined within the record format.

Fields in DISK, or PRINTER file record formats are affected only if the record format is output in the program. Input-only fields are not affected by the RESET operation, except when *ALL is specified.

For more information, see "Initialization Operations" on page 366.

Note: Input-only fields in logical files will appear in the output specifications, although they are not actually written to the file. When a CLEAR or RESET without *NOKEY being specified is done to a record containing these fields, then these fields will be cleared or reset because they appear in the output specifications.

Clearing Entry Fields on a Window

If the *name* operand specifies a window, the window must contain entry fields.

All entry fields on the window are cleared to their default values:
- Numeric fields are cleared with zeros
- Character fields are cleared with blanks.

The corresponding program fields are also set to zero or blank, depending on their type. For example, if window INVENTORY contains the character entry field ENT0000B and the numeric entry field ENT0000C, the CLEAR operation performs the equivalent to the following:

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
                    EVAL      ENT0000B = *BLANKS
                    EVAL      ENT0000C = *ZERO
                    EVAL      %setatr('inventory':'ent0000b':'text') = ENT0000B
                    EVAL      %setatr('inventory':'ent0000c':'text') = ent0000c
```

Figure 233. Clearing windows

Clearing Subfiles

If the *name* operand specifies a subfile, all entries in the subfile are cleared and its Count attribute is set to zero.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D DS1             DS
D  Num                     2      5  0
D  Char                   20     30A
D
D MODS            DS                   OCCURS(2)
D  Fld1                    1      5
D  Fld2                    6     10  0

 *  In the following example, CLEAR sets all subfields in the data
 *  structure DS1 to their defaults, CHAR to blank, NUM to zero.
 /FREE
   CLEAR DS1;

  //  In the following example, CLEAR sets all occurrences for the
  //  multiple occurrence data structure MODS to their default values
  //  Fld1 to blank, Fld2 to zero.
   CLEAR *ALL MODS;
 /END-FREE
```

Figure 234. CLEAR Operation

CLOSE (Close Files)

Free-Form Syntax	CLOSE{(E)} *file-name*|*ALL

Code	Factor 1	Factor 2	Result Field	Indicators		
CLOSE (E)		file-name or *ALL		_	ER	_

The CLOSE operation closes one or more files. The file cannot be used again unless you specify an OPEN operation for that file. The file can either be a local file or a remote file.

A CLOSE operation to an already closed file does not produce an error.

file-name names the file to be closed. You can specify the keyword *ALL to close all the files at once. You cannot specify an array or table file (identified by a T in position 18 of the file description specifications).

To handle CLOSE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

Positions 71, 72, 75, and 76 must be blank.

For more information, see "File Operations" on page 363.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
*  The explicit CLOSE operation closes FILEB.

/FREE
   CLOSE FILEB;

 //  The CLOSE *ALL operation closes all files in the
 //  program. You must specify an explicit OPEN for any file that
 //  you wish to use again.  If the CLOSE operation is not
 //  completed successfully, %ERROR returns '1'.

   CLOSE(E) *ALL;

/END-FREE
```

Figure 235. CLOSE Operation

CLSWIN (Close Window)

Free-Form Syntax	CLSWIN{(E)} *window-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
CLSWIN (E)		Window name		_	ER	_

The CLSWIN operation closes a window and removes it from the display. A Destroy event is generated for the window. The window must be defined in the application.

Factor 2 contains the name of the window to be closed.

To handle CHECKR exceptions, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C* A window named UPDCUST is closed.
C                   CLSWIN    'UPDCUST'
```

Figure 236. CLSWIN Operation

COMMIT (Commit)

Free-Form Syntax	COMMIT{(E)}

Code	Factor 1	Factor 2	Result Field	Indicators		
COMMIT (E)				_	ER	_

The COMMIT operation processes a group of database changes as a unit. Changes associated with the unit can be rolled back using the ROLBK operation.

The COMMIT operation can only be used with OS/400 files. It cannot be used with local files.

To open an OS/400 database file for commitment control, specify COMMIT on the file description specification. Only files opened under commitment control are affected by the COMMIT operation, regardless of the component that issued the COMMIT.

The COMMIT operation does not change the file position. All record locks are released for files under commitment control.

A commitment control environment can only be started for one server. You can use these files on other servers, however these files cannot be operated on under commitment control.

Commitment control ends when the application ends. If changes are pending in the OS/400 database which have not been explicitly committed or rolled back, the changes are rolled back when the application ends. Prior to running an application under a commitment control environment, you must use the GUI Designer to define the commitment level. For more information on using the GUI Designer to define server information, see *Programming with VisualAge RPG* .

To handle COMMIT exceptions (program status codes 802 to 805), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For example, an error occurs if commitment control is not active. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "File Operations" on page 363.

COMP (Compare)

Free-Form Syntax	(not allowed - use the use the =, <, <=, >, >=, or <> operators)

Code	Factor 1	Factor 2	Result Field	Indicators		
COMP	Comparand	Comparand		HI	LO	EQ

The COMP operation compares factor 1 with factor 2.

Factor 1 and factor 2 must contain a literal, a named constant, a field name, a table name, an array element, a data structure, or a figurative constant. Factor 1 and factor 2 must have the same data type. Do not specify the same indicator for all three conditions. When specified, the resulting indicators are set on or off to reflect the results of the compare.

As a result of the comparison, indicators are set on as follows:
- High: (71-72) Factor 1 is greater than factor 2.
- Low: (73-74) Factor 1 is less than factor 2.
- Equal: (75-76) Factor 1 equals factor 2.

"Compare Operations" on page 357 describes the general rules for specifying compare operations.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  Initial field values are:
C*                    FLDA = 100.00
C*                    FLDB = 105.00
C*                    FLDC = 100.00
C*                    FLDD = ABC
C*                    FLDE = ABCDE
C*
C*  Indicator 12 is set on; indicators 11 and 13 are set off.
C     FLDA          COMP      FLDB                                 111213
C*
C*  Indicator 15 is set on; indicator 14 is set off.
C     FLDA          COMP      FLDB                                 141515
C*
C*  Indicator 18 is set on; indicator 17 is set off.
C     FLDA          COMP      FLDC                                 171718
C*
C*  Indicator 21 is set on; indicators 20 and 22 are set off
C     FLDD          COMP      FLDE                                 202122
```

Figure 237. COMP Operation

DEALLOC (Free Storage)

Free-Form Syntax	DEALLOC{(EN)} *pointer-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
DEALLOC (E/N)			pointer-name	_	ER	_

The DEALLOC operation frees one previous allocation of heap storage. *pointer-name* is a pointer that must be the value previously set by a heap-storage allocation operation (either an ALLOC operation in RPG, or some other heap-storage allocation mechanism). It is not sufficient to simply point to heap storage; the pointer must be set to the beginning of an allocation.

The storage pointed to by the pointer is freed for subsequent allocation by this program or any other in the activation group.

If operational extender N is specified, the pointer is set to *NULL after a successful deallocation.

To handle DEALLOC exceptions (program status code 426), either the operation code extender 'E' or an error indicator ER can be specified, but not both. The *pointer-name* operand will not be changed if an error occurs, even if 'N' is specified. For more information on error handling, see "Program Exception and Errors" on page 51.

pointer-name must be a basing pointer scalar variable (a standalone field, data structure subfield, table name or array element).

No error is given at runtime if the pointer is already *NULL.

For more information, see "Memory Management Operations" on page 367.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
 *
D Ptr1            S               *
D Fld1            S              1A
D BasedFld        S              7A   BASED(Ptr1)

 /FREE
    // 7 bytes of storage are allocated from the heap and
    // Ptr1 is set to point to it
    Ptr1 = %alloc (7);

    // The DEALLOC frees the storage.  This storage is now available
    // for allocation by this program or any other program in the
    // activation group.  (Note that the next allocation may or
    // may not get the same storage back).
    dealloc Ptr1;

    // Ptr1 still points at the deallocated storage, but this pointer
    // should not be used with its current value.  Any attempt to
    // access BasedFld which is based on Ptr1 is invalid.
    Ptr1 = %addr (Fld1);

    // The DEALLOC is not valid because the pointer is set to the
    // address of program storage.  %ERROR is set to return '1',
    // the program status is set to 00426 (%STATUS returns 00426),
    // and the pointer is not changed.
    dealloc(e) Ptr1;

    // Allocate and deallocate storage again.  Since operational
    // extender N is specified, Ptr1 has the value *NULL after the
    // DEALLOC.
    Ptr1 = %alloc (7);
    dealloc(n) Ptr1;
 /END-FREE
```

Figure 238. DEALLOC operation

DEFINE (Field Definition)

Free-Form Syntax	(not allowed - use the LIKE or DTAARA keyword on the Definition specification)

Code	Factor 1	Factor 2	Result Field	Indicators		
DEFINE	*LIKE	Referenced field	Defined field			
DEFINE	*DTAARA	External data area	Internal field			

Use the DEFINE operation to either define a field based on the attributes (length and decimal positions) of another field or define a field with an OS/400 data area.

Conditioning indicators (positions 9 through 11) are not permitted.

Defining a Field Based on Another Field

Factor 1 must contain *LIKE.

Factor 2 must contain the name of the field being referenced. This field can be program described or externally described. The attributes of the field in factor 2 are used for the field being defined in the result field. This field can be program described or externally described. Factor 2 cannot be a literal or a named constant, a float numeric field, or an object. If factor 2 is an array, an array element, or a table name, the attributes of an element of the array or table are used to define the field.

The result field contains the name of the field being defined. It cannot be an array, an array element, a data structure, or a table name.

You can use positions 64 through 68 (field length) to make the result field entry longer or shorter than the factor 2 entry. Position 64 can contain either a plus sign (+) to indicate an increase in the field length, or a minus sign (–) to indicate a decrease in the field length. Positions 65 through 68 can contain the increase or decrease in length (right-adjusted) or can be blank. The field length entry is allowed only for graphic, UCS-2, numeric, and character fields. For graphic or UCS-2 fields, the field length difference is calculated in double byte characters.

If positions 64 through 68 are blank, the result field entry is defined with the same length as the factor 2 entry.

Note: You cannot change the number of decimal positions for the field being defined.

If factor 2 is a graphic or UCS-2 field, the result field will be defined as the same type, that is, as graphic or UCS-2. The new field will have the default graphic or UCS-2 CCSID. If you want the new field to have the same CCSID as the field in factor 2, use the LIKE keyword on a definition specification. The length adjustment is expressed in double bytes.

See Figure 239 on page 550 for examples of *LIKE DEFINE.

Defining a Field as a Data Area

Factor 1 must contain *DTAARA.

If factor 2 is specified, it must contain the OS/400 data area being referenced. If factor 2 is not specified, the result field is used as the data area name.

The data area name can either be the OS/400 data area name or an override name you defined using the Define server information menu item. For more information on using the GUI Designer to define server information, see *Programming with VisualAge RPG*.

The result field must contain a field, a data structure, a data structure subfield, or a data area data structure. This is the same name that is used with the IN and OUT operations to retrieve data from and write data to the data area specified in factor 2. When a data area data structure is specified in the result field, the VisualAge RPG application retrieves data from the data area at the program start time and writes data to the data area when the program ends.

The result field cannot contain the following:
- The name of a program status data structure or the name of a subfield of a program status data structure
- A file information data structure or the name of a subfield of a file information data structure
- The name of a subfield of a data area data structure
- A multiple-occurrence data structure or the name of a subfield of a multiple-occurrence data structure
- A data structure that appears in another *DTAARA DEFINE statement
- The data area name on the DTAARA keyword on the definition specification
- An input record field
- An array
- An array element
- A table

Note: If the result field is a data area data structure that contains a packed decimal subfield, the OS/400 data area must contain a valid packed decimal value that has been initialized.

For numeric data areas, the maximum length is 24 digits with 9 decimal places. Note that there is a maximum of 15 digits to the left of the decimal place, even if the number of decimals is less than 9.

You can use positions 64 though 70 to define the length and number of decimal positions for the result field. This must match the external description of the data area specified in factor 2.

See Figure 239 on page 550 for examples of *DTAARA DEFINE.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
*
C*  FLDA is a 7-position character field.
C*  FLDB is a 5-digit field with 2 decimal positions.
C*
C*
C*  FLDP is a 7-position character field.
C     *LIKE         DEFINE    FLDA          FLDP
C*
C*  FLDQ is a 9-position character field.
C     *LIKE         DEFINE    FLDA          FLDQ             +2
C*
C*  FLDR is a 6-position character field.
C     *LIKE         DEFINE    FLDA          FLDR            - 1
C*
C*  FLDS is a 5-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDS
C*
C*  FLDT is a 6-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDT           +  1
C*
C*  FLDU is a 3-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDU            - 2
C*
C*  FLDX is a 3-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDU          FLDX
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The attributes (length and decimal positions) of
C*  the data area (TOTGRS) must be the same as those for the
C*  external data area.
C
C     *DTAARA       DEFINE                  TOTGRS           10 2
C
C*
C*  The result field entry (TOTNET) is the name of the data area to
C*  be used within the VRPG program.  The factor 2 entry (TOTAL)
C*  is the name of the data area as defined to the system.
C
C     *DTAARA       DEFINE    TOTAL         TOTNET
C
```

Figure 239. DEFINE Operation

DELETE (Delete Record)

Free-Form Syntax	DELETE{(EHMR)} {*search-arg*} *name*

Code	Factor 1	Factor 2	Result Field	Indicators		
DELETE (E)	search-arg	name (file, record format, or subfile)		NR	ER	_

The DELETE operation deletes a record. Once the record has been deleted, it can never be retrieved.

If a search argument (*search-arg*) is not specified, the DELETE operation deletes the current record. The current record is the last record retrieved. The record must be locked by a previous input operation such as CHAIN or READ.

If a search argument (*search-arg*) is specified, it must contain a key, relative record number, or a subfile index number that identifies the record to be deleted:

- If access is by key, the *name* operand must be a remote file. If duplicate records exist for the key, only the first of the duplicate records is deleted from the file.

  The *search-arg* can be a single key in the form of a field name, a named constant, a figurative constant, or a literal.

  If the file is an externally-described file, *search-arg* can also be a composite key in the form of a KLIST name, a list of values, or %KDS. Graphic and UCS-2 key fields must have the same CCSID as the key in the file. For an example of %KDS, see the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451. For an example of using a list of values to search for the record to be deleted, see Figure 227 on page 531.
- If access is by relative record number or subfile index number, the search argument must be a numeric constant or variable with zero decimal positions.
- Graphic and UCS-2 key fields must have the same CCSID as the key in the file.

The *name* operand must be the name of the file or the name of a record format in the file from which a record is to be deleted:

- The file can either be an OS/400 file or a local file.
- A record format name can only be used with an externally described OS/400 file. If a search argument is not specified, the record format name must be the name of the last record read from the file; otherwise, an error occurs.

If the search argument is specified, positions 71 and 72 can contain an indicator that is set on if the record to be deleted is not found in the file. If the search argument is not specified, leave these positions blank. This information can also be obtained from the %FOUND built-in function, which returns '0' if no record is found, and '1' if a record is found.

To handle DELETE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

Note the following when deleting records:

- Deleting a record from a subfile part causes a shift in the subfile record index numbers among the remaining records in the subfile.
- If a sequential read operation is done on the file after a successful DELETE operation to that file, the next record after the deleted record is obtained.

For more information, see “File Operations” on page 363.

Notes:

1. Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().
2. Leave positions 75 and 76 blank.

DIV (Divide)

Free-Form Syntax	(not allowed - use the / or /= operator, or the%DIV built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
DIV (H)	Dividend	Divisor	Quotient	+	–	Z

If factor 1 is specified, the DIV operation divides factor 1 by factor 2; otherwise it divides the result field by factor 2. The quotient is placed in the result field. If factor 1 is 0, the result of the operation is 0. Factor 2 cannot be zero. If it is, the VARPG exception/error handling routine receives control. Factor 1 and factor 2 must be numeric; each can contain one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

Factor 2 cannot be 0. If it is, the VRPG Client exception/error handling routine receives control. Factor 2 must be numeric and can contain an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

Any remainder resulting from the divide operation is lost unless the move remainder (MVR) operation is specified as the next operation. If you use conditioning indicators, the DIV operation must be specified immediately before the MVR operation.

The result of the divide operation cannot be half-adjusted (rounded) if the MVR operation is specified after the DIV operation.

Note: The MVR operation cannot follow a DIV operation if any operand of the DIV operation is of float format. A float variable can, however, be specified as the result of operation code MVR.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the DIV operation.

DO (Do)

Free-Form Syntax	(not allowed - use the FOR operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
DO	Starting value	Limit value	Index value			

The DO operation begins a group of operations and indicates the number of times the group will be processed. To indicate the number of times the group of operations is to be processed, specify an index field, a starting value, and a limit value. An associated ENDDO statement marks the end of the group. For more information on DO groups, see "Structured Programming Operations" on page 376.

If factor 1 is specified, it must contain a numeric literal, named constant, or field name. If factor 1 is not specified, the starting value is 1.

If factor 2 is specified, it must contain a numeric field name, literal, or named constant. Factor 2 must be specified with zero decimal positions, If factor 2 is not specified, the limit value is 1.

If the result field is specified, it must be a numeric field name that is large enough to contain the limit value plus the increment. Any value in the result field is replaced by factor 1 when the DO operation begins.

Factor 2 of the associated ENDDO operation specifies the value to be added to the index field (the result field of the DO operation). It must be a numeric literal or a numeric field with no decimal positions. If it is not specified, 1 is added to the index field.

In addition to the DO operation itself, the conditioning indicators on the DO and ENDDO statements control the DO group. The conditioning indicators on the DO statement control whether or not the DO operation begins. These indicators are checked only once, at the beginning of the DO loop. The conditioning indicators on the associated ENDDO statement control whether or not the DO group is repeated another time. These indicators are checked at the end of each loop.

The DO operation follows these 7 steps:

1. If the conditioning indicators on the DO statement line are satisfied, the DO operation is processed (step 2). If the indicators are not satisfied, control passes to the next operation to be processed following the associated ENDDO statement (step 7).
2. The starting value (factor 1) is moved to the index field (result field) when the DO operation begins.
3. If the index value is greater than the limit value, control passes to the calculation operation following the associated ENDDO statement (step 7). Otherwise, control passes to the first operation after the DO statement (step 4).
4. Each of the operations in the DO group is processed.
5. If the conditioning indicators on the ENDDO statement are not satisfied, control passes to the calculation operation following the associated ENDDO statement (step 7). Otherwise, the ENDDO operation is processed (step 6).

6. The ENDDO operation is processed by adding the increment to the index field. Control passes to step 3. (Note that the conditioning indicators on the DO statement are not tested again (step 1) when control passes to step 3.)
7. The statement after the ENDDO statement is processed when the conditioning indicators on the DO or ENDDO statements are not satisfied (step 1 or 5), or when the index value is greater than the limit value (step 3).

Note: The index, increment, limit value, and indicators can be modified within the loop to affect the ending of the DO group.

See "LEAVE (Leave a Do/For Group)" on page 596 and "ITER (Iterate)" on page 591 for a description of how those operations affect a DO operation.

See "FOR (For)" on page 581 for information on performing iterative loops with **free-form expressions** for the initial, increment, and limit values.

For more information, see "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The DO group is processed 10 times when indicator 17 is on;
C*  it stops running when the index value in field X, the result
C*  field, is greater than the limit value (10) in factor 2.  When
C*  the DO group stops running, control passes to the operation
C*  immediately following the ENDDO operation.  Because factor 1
C*  in the DO operation is not specified, the starting value is 1.
C*  Because factor 2 of the ENDDO operation is not specified, the
C*  incrementing value is 1.
C
C   17              DO        10            X                 3 0
C                   :
C                   ENDDO
C*
C*  The DO group can be processed 10 times.  The DO group stops
C*  running when the index value in field X is greater than
C*  the limit value (20) in factor 2, or if indicator 50 is not on
C*  when the ENDDO operation is encountered.  When indicator 50
C*  is not on, the ENDDO operation is not processed; therefore,
C*  control passes to the operation following the ENDDO operation.
C*  The starting value of 2 is specified in factor 1 of the DO
C*  operation, and the incrementing value of 2 is specified in
C*  factor 2 of the ENDDO operation.
C*
C     2             DO        20            X                 3 0
C                   :
C                   :
C                   :
C   50              ENDDO     2
```

Figure 240. DO Operation

DOU (Do Until)

Free-Form Syntax	DOU{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2
DOU (M/R)		indicator-expression

The DOU operation is similar to the DOUxx operation. The DOU operation code precedes a group of operations which you want to execute at least once and possibly more than once. An associated ENDDO statement marks the end of the group. It differs in that the logical condition is expressed by an indicator valued expression (*indicator-expression*).

The operations controlled by the DOU operation are performed until the expression in (*indicator-expression*) is true. For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 390.

For fixed-format syntax, level and conditioning indicators are valid. Factor 1 must be blank. Extended factor 2 contains the expression to be evaluated.

Chapter 24, "Expressions," on page 381 describes how to specify expressions.

"Compare Operations" on page 357 describes the rules for specifying the compare operations.

For more information, see "Compare Operations" on page 357 or "Structured Programming Operations" on page 376.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
   // In this example, the do loop will be repeated until the F3
   // is pressed.
   dou *inkc;
      do_something();
   enddo;

   // The following do loop will be repeated until *In01 is on
   // or until FIELD2 is greater than FIELD3
   dou *in01 or (Field2 > Field3);
      do_something_else ();
   enddo;

   // The following loop will be repeated until X is greater than
   // the number of elements in Array
   dou X > %elem (Array);
      Total = Total + Array(x);
      X = X + 1;
   enddo;
/END-FREE
```

Figure 241. DOU Operation

DOUxx (Do Until)

Free-Form Syntax	(not allowed - use the DOU operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
DOUxx	Comparand	Comparand				

The DOUxx operation code precedes a group of operations which you want to execute at least once and possibly more than once. An associated ENDDO statement marks the end of the group. For more information on DO groups and the meaning of xx, see "Structured Programming Operations" on page 376.

Factor 1 and factor 2 must contain a literal, a named constant, a field name, a table name, an array element, a figurative constant, or a data structure name. Factor 1 and factor 2 must be the same data type.

On the DOUxx statement, you indicate a relationship xx. To specify a more complex condition, immediately follow the DOUxx statement with ANDxx or ORxx statements. The operations in the DO group are processed once, and then the group is repeated while the relationship exists between factor 1 and factor 2 or the condition specified by a combined DOUxx, ANDxx, or ORxx operation exists. The group is always processed at least once even if the condition is not true at the start of the group.

In addition to the DOUxx operation itself, the conditioning indicators on the DOUxx and ENDDO statements control the DO group. The conditioning indicators on the DOUxx statement control whether or not the DOUxx operation begins. The conditioning indicators on the associated ENDDO statement can cause a DO loop to end prematurely.

The DOUxx operation follows these steps:

1. If the conditioning indicators on the DOUxx statement line are satisfied, the DOUxx operation is processed (step 2). If the indicators are not satisfied, control passes to the next operation that can be processed following the associated ENDDO statement (step 6).
2. The DOUxx operation is processed by passing control to the next operation that can be processed (step 3). The DOUxx operation does not compare factor 1 and factor 2 or test the specified condition at this point.
3. Each of the operations in the DO group is processed.
4. If the conditioning indicators on the ENDDO statement are not satisfied, control passes to the next calculation operation following the associated ENDDO statement (step 6). Otherwise, the ENDDO operation is processed (step 5).
5. The ENDDO operation is processed by comparing factor 1 and factor 2 of the DOUxx operation or testing the condition specified by a combined operation. If the relationship xx exists between factor 1 and factor 2 or the specified condition exists, the DO group is finished and control passes to the next calculation operation after the ENDDO statement (step 6). If the relationship xx does not exist between factor 1 and factor 2 or the specified condition does not exist, the operations in the DO group are repeated (step 3).

6. The statement after the ENDDO statement is processed when the conditioning indicators on the DOUxx or ENDDO statements are not satisfied (steps 1 or 4), or when the relationship xx between factor 1 and factor 2 or the specified condition exists at step 5.

See "LEAVE (Leave a Do/For Group)" on page 596 and "ITER (Iterate)" on page 591 for information on how those operations affect a DOUxx operation.

For more information, see "Compare Operations" on page 357 or "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The DOUEQ operation runs the operation within the DO group at
C*  least once.
C
C     FLDA          DOUEQ     FLDB
C
C*
C*  At the ENDDO operation, a test is performed to determine whether
C*  FLDA is equal to FLDB.  If FLDA does not equal FLDB, the
C*  preceding operations are processed again.  This loop continues
C*  processing until FLDA is equal to FLDB.  When FLDA is equal to
C*  FLDB, the program branches to the operation immediately
C*  following the ENDDO operation.
C
C                   SUB       1             FLDA
C                   ENDDO
C
C*
C*  The combined DOUEQ ANDEQ OREQ operation processes the operation
C*  within the DO group at least once.
C
C     FLDA          DOUEQ     FLDB
C     FLDC          ANDEQ     FLDD
C     FLDE          OREQ      100
C
C*
C*  At the ENDDO operation, a test is processed to determine whether
C*  the specified condition, FLDA equal to FLDB and FLDC equal to
C*  FLDD, exists.  If the condition exists, the program branches to
C*  the operation immediately following the ENDDO operation.  There
C*  is no need to test the OREQ condition, FLDE equal to 100, if the
C*  DOUEQ and ANDEQ conditions are met.  If the specified condition
C*  does not exist, the OREQ condition is tested.  If the OREQ
C*  condition is met, the program branches to the operation
C*  immediately following the ENDDO.  Otherwise, the operations
C*  following the OREQ operation are processed and then the program
C*  processes the conditional tests starting at the second DOUEQ
C*  operation.  If neither the DOUEQ and ANDEQ condition nor the
C*  OREQ condition is met, the operations following the OREQ
C*  operation are processed again.
C
C                   SUB       1             FLDA
C                   ADD       1             FLDC
C                   ADD       5             FLDE
C                   ENDDO
```

Figure 242. DOUxx Operations

DOW (Do While)

Free-Form Syntax	DOW{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2
DOW (M/R)		indicator-expression

The DOW operation code precedes a group of operations which you want to process when a given condition exists. An associated ENDDO statement marks the end of the group. Its function is similar to that of the DOWxx operation code. It differs in that the logical condition is expressed by an indicator valued expression (*indicator-expression*). The operations controlled by the DOW operation are performed while the expression in *indicator-expression* is true. For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 390.

For fixed-format syntax, level and conditioning indicators are valid. Factor 1 must be blank. Factor 2 contains the expression to be evaluated.

"Compare Operations" on page 357 describes the rules for specifying the compare operations.

For more information, see "Compare Operations" on page 357 or "Structured Programming Operations" on page 376.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
* In this example, the do loop will be repeated until the condition
* is false. That is when A > 5 or B+C are not equal to zero.

/FREE
   dow (a <= 5) and (b + c = 0);
      do_something (a:b:c);
   enddo;
/END-FREE
```

Figure 243. DOW Operation

DOWxx (Do While)

Free-Form Syntax	(not allowed - use the DOW operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
DOWxx	Comparand	Comparand				

The DOWxx operation code precedes a group of operations which you want to process when a given condition exists. To specify a more complex condition, immediately follow the DOWxx statement with ANDxx or ORxx statements. An associated ENDDO statement marks the end of the group. For further information on DO groups and the meaning of xx, see "Structured Programming Operations" on page 376.

Factor 1 and factor 2 must contain a literal, a named constant, a figurative constant, a field name, a table name, an array element, or a data structure name. Factor 1 and factor 2 must be of the same data type. The comparison of factor 1 and factor 2 follows the same rules as those given for the compare operations. See "Compare Operations" on page 357.

In addition to the DOWxx operation itself, the conditioning indicators on the DOWxx and ENDDO statements control the DO group. The conditioning indicators on the DOWxx statement control whether or not the DOWxx operation is begun. The conditioning indicators on the associated ENDDO statement control whether the DOW group is repeated another time.

The DOWxx operation follows these steps:

1. If the conditioning indicators on the DOWxx statement line are satisfied, the DOWxx operation is processed (step 2). If the indicators are not satisfied, control passes to the next operation to be processed following the associated ENDDO statement (step 6).
2. The DOWxx operation is processed by comparing factor 1 and factor 2 or testing the condition specified by a combined DOWxx, ANDxx, or ORxx operation. If the relationship xx between factor 1 and factor 2 or the condition specified by a combined operation does not exist, the DO group is finished and control passes to the next calculation operation after the ENDDO statement (step 6). If the relationship xx between factor 1 and factor 2 or the condition specified by a combined operation exists, the operations in the DO group are repeated (step 3).
3. Each of the operations in the DO group is processed.
4. If the conditioning indicators on the ENDDO statement are not satisfied, control passes to the next operation to run following the associated ENDDO statement (step 6). Otherwise, the ENDDO operation is processed (step 5).
5. The ENDDO operation is processed by passing control to the DOWxx operation (step 2). (Note that the conditioning indicators on the DOWxx statement are not tested again at step 1.)
6. The statement after the ENDDO statement is processed when the conditioning indicators on the DOWxx or ENDDO statements are not satisfied (steps 1 or 4), or when the relationship xx between factor 1 and factor 2 of the specified condition does not exist at step 2.

See "LEAVE (Leave a Do/For Group)" on page 596 and "ITER (Iterate)" on page 591 for information on how those operations affect a DOWxx operation.

For more information, see "Compare Operations" on page 357 or "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The DOWLT operation allows the operation within the DO group
C*  to be processed only if FLDA is less than FLDB.  If FLDA is
C*  not less than FLDB, the program branches to the operation
C*  immediately following the ENDDO operation.  If FLDA is less
C*  than FLDB, the operation within the DO group is processed.
C
C     FLDA          DOWLT     FLDB
C
C*
C*  The ENDDO operation causes the program to branch to the first
C*  DOWLT operation where a test is made to determine whether FLDA
C*  is less than FLDB.  This loop continues processing until FLDA
C*  is equal to or greater than FLDB; then the program branches
C*  to the operation immediately following the ENDDO operation.
C
C                   MULT      2.08          FLDA
C                   ENDDO
C
C*  In this example, multiple conditions are tested.  The combined
C*  DOWLT ORLT operation allows the operation within the DO group
C*  to be processed only while FLDA is less than FLDB or FLDC. If
C*  neither specified condition exists, the program branches to
C*  the operation immediately following the ENDDO operation. If
C*  either of the specified conditions exists, the operation after
C*  the ORLT operation is processed.
C
C     FLDA          DOWLT     FLDB
C     FLDA          ORLT      FLDC
C
C*  The ENDDO operation causes the program to branch to the second
C*  DOWLT operation where a test determines whether specified
C*  conditions exist.  This loop continues until FLDA is equal to
C*  or greater than FLDB and FLDC; then the program branches to the
C*  operation immediately following the ENDDO operation.
C
C                   MULT      2.08          FLDA
C                   ENDDO
```

Figure 244. DOWxx Operations

DSPLY (Display Message Window)

Free-Form Syntax	DSPLY{(E)} *message* {*message-window-definition-name*	*DFT {*response*}}

Code	Factor 1	Factor 2	Result Field	Indicators		
DSPLY (E)	message	message-window-definition-name	response	_	ER	_

The DSPLY operation displays a Message window. The program halts, displays the message window, and waits for a response.

The *message* operand must be one of the following:
- A field name
- A field name defined on the MSGNBR keyword
- Character literal, numeric literal, hexadecimal literal, DBCS literal, date literal, time literal, or timestamp literal
- A Definition specification name
- Event attributes
- Message identifier

The *message* operand can be an event attribute provided the DSPLY operation is in an action subroutine for an appropriate event (that is, the event has the specified event attribute) or the DSPLY operation is a user subroutine executed by an action subroutine for an event. Within free-form calculations, the message operand can be an expression, provided the expression is enclosed by parentheses. If the EXE or NOMAIN keyword is specified on a control specification, you cannot use a Message Box description or message identifier as a field name.

Pointer fields are not allowed. Except for message numbers, all data in *message* is converted to character before being displayed.

If the *message-window-definition-name* is specified, it must contain the Definition specification name that defines the style. *message-window-definition-name* is optional when:
- *message* is a message identifier (*MSGnnnn)
- *message* is a Definition specification name and the referenced Definition specification contains the MSGNBR keyword. The MSGNBR can be either the message number or a field containing the message number.

message-window-definition-name is ignored if the EXE or NOMAIN keyword is specified on a control specification.

In free-form syntax, *DFT can be specified as a placeholder for no *message-window-definition-name* operand in order to specify the *response* operand.

If specified, the *response* operand receives a value representing the button in the Message window that the user pressed. The value corresponds to one of the figurative constants that are used to define which buttons appear in a Message Box (for example, *RETRY). These constants should be used to check which button the user pressed. The *response* operand must be a numeric field with a length of 9 and no decimal positions.

If the EXE or NOMAIN keyword is specified, the *response* operand can be numeric with precision 9,0 or character. Reply fields in Message windows behave differently for numeric and character fields. Numeric reply fields behave as follows:
- Pressing Enter or Return in the field returns the value 0.
- The field accepts only 9 digits; if more than 9 are entered, they are ignored.
- Entering a character including the decimal point causes a runtime error and ends the program.

Character fields behave as follows:
- Pressing Enter or Return fills the field with blanks.
- Extra characters typed in the field are ignored. The field can accept one or more words.

When the NOMAIN keyword is used, procedures called from Windows GUI applications behave as follows:
- For character or DBCS fields, no message is displayed and the reply field is filled with blanks.
- Numeric fields are filled the value 0.

To handle DSPLY exceptions, either the operation code extender 'E' or an error indicator ER can be specified, but not both. The exception is handled by the specified method if an error occurs on the operation. For more information on error handling, see "Program Exception and Errors" on page 51.

Various keywords in the Definition specification are used to define the Message window. The BUTTON, MSGTITLE, and STYLE keywords define the window style. The MSGDATA, MSGNBR, and MSGTEXT keywords define the message text that appears in the window. Refer to "Definition-Specification Keywords" on page 264.

For more information, see "Message Operations" on page 368.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D Box1            M                   STYLE(*WARN)  BUTTON(*RETRY:*ABORT:*IGNORE)
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C     *MSG9999      DSPLY     BOX1          REPLY             9 0
C                   IF        reply = *RETRY
 *  Retry button was pressed
C                   .....
C                   ELSE
C                   IF        reply = *ABORT
 *  Abort button was pressed
C                   .....
C                   ELSE
 *  Ignore button was pressed
C                   .....
C                   ENDIF
C                   ENDIF
```

Figure 245. DSPLY Operation

ELSE (Else)

Free-Form Syntax	ELSE

Code	Factor 1	Factor 2	Result Field	Indicators		
ELSE						

The ELSE operation is an optional part of the IFxx and IF operations. If the IFxx comparison is met, the calculations before ELSE are processed; otherwise, the calculations after ELSE are processed.

Conditioning indicator entries (positions 9 through 11) are not permitted.

To close the IFxx/ELSE group use an ENDIF operation.

Figure 258 on page 588 shows an example of an ELSE operation with an IFxx operation.

For more information, see “Structured Programming Operations” on page 376.

ELSEIF (Else If)

Free-Form Syntax	ELSEIF{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2
ELSEIF (M/R)	Blank	indicator-expression

The ELSEIF operation is the combination of an ELSE operation and an IF operation. It avoids the need for an additional level of nesting.

The IF operation code allows a series of operation codes to be processed if a condition is met. Its function is similar to that of the IFxx operation code. It differs in that the logical condition is expressed by an indicator valued expression (*indicator-expression*). The operations controlled by the ELSEIF operation are performed when the expression in the *indicator-expression* operand is true (and the expression for the previous IF or ELSEIF statement was false).

For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 390.

For more information, see "Structured Programming Operations" on page 376.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/free

   IF state = 0;
      dosomething();
   ELSEIF state = 1;
      return;
   ELSEIF state = 2;
      report(state);
   ELSE;
      signalError ('Bad state');
   ENDIF;

/end-free
```

Figure 246. ELSEIF Operation

ENDyy (End a Structured Group)

Free-Form Syntax	ENDDO
	ENDFOR
	ENDIF
	ENDMON
	ENDSL
	(END and ENDCS not allowed)

Code	Factor 1	Factor 2	Result Field	Indicators		
END		**increment-value**				
ENDCS						
ENDDO		increment-value				
ENDFOR						
ENDIF						
ENDMON						
ENDSL						

The ENDyy operation ends a CASxx, DO, DOU, DOW, DOUxx, DOWxx, FOR, IF, IFxx, MONITOR,or SELECT group of operations.

The ENDyy operation ends a CASxx, DO, DOU, DOW, DOUxx, DOWxx, FOR, IF, IFxx, or SELECT group of operations.

The ENDyy operations are listed below:

END End a CASxx, DO, DOU, DOUxx, DOW, DOWxx, FOR, IF, IFxx, or SELECT group

ENDCS End a CASxx group

ENDDO End a DO, DOU, DOUxx, DOW, or DOWxx group

ENDFOR End a FOR group

*** Restriction:** ENDFOR is unsupported in Java applications.

ENDIF End an IF or IFxx group

ENDMON End a MONITOR group

ENDSL End a SELECT group

The *increment-value* operand is allowed only on an ENDyy operation that delimits a DO group. It contains the incrementing value of the DO group. It can be positive or negative, must have zero decimal positions, and can be an array element, table name, data structure, field, named constant, or numeric literal. If *increment-value* is not specified, the increment defaults to 1. If *increment-value* is negative, the DO group never ends.

Conditioning indicators can be specified for an ENDDO or ENDFOR operation. They are not allowed for ENDCS, ENDIF, ENDMON, and ENDSL.

Resulting indicators are not allowed. No operands are allowed for ENDCS, ENDIF, ENDMON, ENDSL.

If one ENDyy form is used with a different operation group (for example, ENDIF with a structured group), an error results at compilation time.

For more information, see the following for examples that use the ENDyy operation:
- "CASxx (Conditionally Invoke Subroutine)" on page 524
- "DO (Do)" on page 554
- "DOUxx (Do Until)" on page 557
- "DOWxx (Do While)" on page 560
- "IFxx (If)" on page 587
- "DOU (Do Until)" on page 556
- "DOW (Do While)" on page 559
- "FOR (For)" on page 581
- "IF (If)" on page 586
- "MONITOR (Begin a Monitor Group)" on page 602
- "SELECT (Begin a Select Group)" on page 676

For more information, see "Error-Handling Operations" on page 362 or "Structured Programming Operations" on page 376.

ENDACT (End of Action Subroutine)

Free-Form Syntax	ENDACT {*return-point*}

Code	Factor 1	Factor 2	Result Field	Indicators		
ENDACT		Return point				

The ENDACT operation defines the end of an action subroutine. ENDACT must be the last operation in an action subroutine.

If factor 2 is specified, it must contain one of the following:

***DEFAULT**
The current action subroutine ends and the default processing for the current event is performed.

***NODEFAULT**
The current action subroutine ends and the default processing for the current event is NOT performed.

a field name
The field name must be 12 characters and can contain *DEFAULT or *NODEFAULT If the field contains an invalid value, the default error handler receives control.

If factor 2 is not specified, the current action subroutine ends and the default processing for the current event is performed.

When processing reaches the ENDACT operation, and if LR is on, the component is terminated. *DEFAULT and *NODEFAULT are ignored. If action subroutines are nested, LR is not checked, and *DEFAULT and *NODEFAULT are ignored.

Conditioning indicator entries are not allowed.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C                   ENDACT    '*DEFAULT'
```

Figure 247. ENDACT Operation

ENDSR (End of User Subroutine)

Free-Form Syntax	ENDSR {*return-point*}

Code	Factor 1	Factor 2	Result Field	Indicators		
ENDSR	label	return-point				

The ENDSR operation defines the end of a user subroutine. It causes a return to the statement following the EXSR operation. ENDSR must be the last operation in the subroutine.

In fixed-format syntax, the *label* operand can be specified as a point to which a GOTO operation within the subroutine can branch. (You cannot specify a *label* in free-form syntax.)

The (*return-point*) operand can only be specified at the end of a *PSSR or *INFSR subroutine. It must contain one of the following:

***CANCL**
The action subroutine that was running when the error occurs finishes and the component ends abnormally.

***ENDCOMP**
The action subroutine that was running when the error occurs finishes and the component ends abnormally.

***DEFAULT**
Control returns from the current action subroutine and the default processing for the current event is performed. If LR is on, the component terminates normally. If LR is not on, the action subroutine ends and any default action for the event is performed.

***NODEFAULT**
Control returns from the current action subroutine and the default processing for the current event is not performed. If LR is on, the component terminates normally. If LR is NOT on, the action subroutine ends and any default action for the event is NOT performed.

***ENDAPPL**
The action subroutine that was running when the error occurs finishes and all currently active components end in reverse hierarchical order. The component that was active when the error occurred terminates normally and all other components terminate normally.

a field name
A field name can contain *CANCL, *ENDCOMP, *DEFAULT, *NODEFAULT, or *ENDAPPL. If the field contains an invalid value, the default error handler receives control.

Conditioning indicators are not allowed.

For more information, see “Subroutine Operations” on page 378.

ENDSR (End of User Subroutine)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                             Extended-factor2++++++++++++++++++++++++++++
C*
C     Label          BEGSR
C                      .
C                      .
C                      .
C                    ENDSR     '*ENDCOMP'
```

Figure 248. ENDSR Operation

EVAL (Evaluate Expression)

Free-Form Syntax	{EVAL{(HMR)}} *result = expression*
	{EVAL{(HMR)}} *result += expression*
	{EVAL{(HMR)}} *result -= expression*
	{EVAL{(HMR)}} *result *= expression*
	{EVAL{(HMR)}} *result /= expression*
	{EVAL{(HMR)}} *result **= expression*

Code	Factor 1	Extended Factor 2
EVAL (H/M/R)		Assignment Statement

The EVAL operation code evaluates an assignment statement of the form rev="v5r2">`"result = expression"` or `"result op = expression"`. The expression is evaluated and the result placed in **result**. Therefore, **result** cannot be a literal or constant but must be a field name, array name, array element, data structure, data structure subfield, or a string using the %SUBST built-in function.

The expression may yield any of the RPG data types. The type of the expression must be the same as the type of the result. A character, graphic, or UCS-2 result will be left justified and padded with blanks on the right or truncated as required. If **result** is a variable-length field, its length will be set to the length of the result of the expression.

If the result represents an unindexed array or an array specified as array(*), the value of the expression is assigned to each element of the result, according to the rules described in "Specifying an Array in Calculations" on page 183. Otherwise, the expression is evaluated once and the value is placed into each element of the array or sub-array. For numeric expressions, the half-adjust operation code extender is allowed. The rules for half adjusting are equivalent to those for the arithmetic operations.

On a free-form calculation specification, the operation code name may be omitted if no extenders are needed.

For the assignment operators +=, -=, *=, /=, and **=, the appropriate operation is applied to the result and the expression, and the result is assigned to the result. For example, statement X+=Y is roughly equivalent to X=X+Y. The difference between the two statements is that for these assignment operators, the result operand is evaluated only once. This difference is significant when the evaluation of the result operation involves a call to a subprocedure which has side-effects, for example:

```
warnings(getNextCustId(OVERDRAWN)) += 1;
```

See Chapter 24, "Expressions," on page 381 for general information on expressions. See "Precision Rules for Numeric Operations" on page 390 for information on precision rules for numeric expressions. This is especially important if the expression contains any divide operations, or if the EVAL uses any of the operation extenders.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
*                   Assume FIELD1 = 10
*                          FIELD2 =  9
*                          FIELD3 =  8
*                          FIELD4 =  7
*                          ARR is defined with DIM(10)
*                          *IN01 = *ON
*                          A = 'abcdefghijklmno' (define as 15 long)
*                          CHARFIELD1 = 'There'  (define as  5 long)

/FREE
  // The content of RESULT after the operation is 20
  eval RESULT=FIELD1 + FIELD2+(FIELD3-FIELD4);
  // The indicator *IN03  will be set to *ON
  *IN03 = *IN01 OR (FIELD2 > FIELD3);
  //  Each element of array ARR will be assigned the value 72
  ARR(*) = FIELD2 * FIELD3;
  //  After the operation, the content of A = 'Hello There   '
  A = 'Hello ' + CHARFIELD1;
  //  After the operation the content of A = 'HelloThere    '
  A = %TRIMR('Hello ') + %TRIML(CHARFIELD1);
  //  Date in assignment
  ISODATE = DMYDATE;
  //  Relational expression
  //  After the operation the value of *IN03 = *ON
  *IN03 = FIELD3 < FIELD2;
  //  Date in Relational expression
  // After the operation, *IN05 will be set to *ON if Date1 represents
  // a date that is later that the date in Date2
  *IN05 = Date1 > Date2;
  // After the EVAL the original value of A contains 'ab****ghijklmno'
  %SUBST(A(3:4))= '****';
  // After the EVAL PTR has the address of variable CHARFIELD1
  PTR = %ADDR(CHARFIELD1);
  //  An example to show that the result of a logical expression is
  //  compatible with the character data type.
  //  The following EVAL statement consisting of 3 logical expressions
  //  whose results are concatenated using the '+' operator
  //  The resulting value of the character field RES is '010'
  RES = (FIELD1<10) + *in01 + (field2 >= 17);
  // An example of calling a user-defined function using EVAL.
  // The procedure FormatDate converts a date field into a character
  // string, and returns that string.  In this EVAL statement, the
  // field DateStrng1 is assigned the output of formatdate.
  DateStrng1 = FormatDate(Date1);
  // Subtract value in complex data structure.
  cust(custno).account(accnum).balance -= purchase_amount;
  // Add days and months to a date
  DATE += %DAYS(12) + %MONTHS(3);
  // Append characters to varying length character variable
  line += '<br />';
/END-FREE
```

Figure 249. EVAL Operations

EVALR (Evaluate expression, right adjust)

Free-Form Syntax	EVALR{(MR)} *result = expression*

Code	Factor 1	Extended Factor 2
EVALR (M/R)		Assignment Statement

The EVALR operation code evaluates an assignment statement of the form `result=expression`. The expression is evaluated and the result is placed right-adjusted in the result. Therefore, the result cannot be a literal or constant, but must be a fixed-length character, graphic, or UCS-2 field name, array name, array element, data structure, data structure subfield, or a string using the %SUBST built-in function. The type of the expression must be the same as the type of the result. The result will be right justified and padded with blanks on the left, or truncated on the left as required.

Notes:

1. Unlike the EVAL operation, the result of EVALR can only be of type character, graphic, or UCS-2. In addition, only fixed length result fields are allowed, although %SUBST can contain a variable length field if this built-in function forms the lefthand part of the expression.
2. EVALR used with the %SETATR or %GETATR built-in behaves like the EVAL operation. There is no right justification of the attribute value when set or retrieved.

If the result represents an unindexed array or an array specified as array(*), the value of the expression is assigned to each element of the result, according to the rules described in “Specifying an Array in Calculations” on page 183. Otherwise, the expression is evaluated once and the value is placed into each element of the array or sub-array.

See Chapter 24, “Expressions,” on page 381 for general information on expressions. See “Precision Rules for Numeric Operations” on page 390 for information on precision rules for numeric expressions. This is especially important if the expression contains any divide operations, or if the EVALR uses any of the operation extenders.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++
D Name            S             20A

 /FREE
    eval  Name = 'Kurt Weill';
    // Name is now 'Kurt Weill          '
    evalr Name = 'Johann Strauss';
    // Name is now '      Johann Strauss'
    evalr %SUBST(Name:1:12) = 'Richard';
    // Name is now '     Richard Strauss'
    eval  Name = 'Wolfgang Amadeus Mozart';
    // Name is now 'Wolfgang Amadeus Moz'
    evalr Name = 'Wolfgang Amadeus Mozart';
    // Name is now 'fgang Amadeus Mozart'
 /END-FREE
```

Figure 250. EVALR Operations

EXCEPT (Calculation Time Output)

Free-Form Syntax	EXCEPT {*except-name*}

Code	Factor 1	Factor 2	Result Field	Indicators		
EXCEPT		except-name				

The EXCEPT operation allows one or more exception records to be written during calculation time. The file the records are written to can either be a local file or an OS/400 file.

The exception records that are to be written during calculation time are indicated by an E in (position 17) on the output specifications. The *except-name* operand must be the same name as the EXCEPT name on the output specifications (positions 30-39) of the exception records.

If a *except-name* operand is specified, only those exception records with the same EXCEPT name are checked and written if the conditioning indicators are satisfied. When no *except-name* is specified, only those exception records on the output specifications (positions 30-39) are checked and written if the conditioning indicators are satisfied.

If an exception output is specified to a format that contains no fields, the following occurs:

- If an output file is specified, a record is written with default values.
- If a record is locked, the system treats the operation as a request to unlock the record. This is the alternative form of requesting an unlock. The preferred method is with the UNLOCK operation.

For more information, see “File Operations” on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  When the EXCEPT operation with HDG specified in factor 2 is
C*  processed, all exception records with the EXCEPT name HDG are
C*  written.  In this example, UDATE and PAGE would be printed
C*  and then the printer would space 2 lines.
C*  The second HDG record would print a line of dots and then the
C*  printer would space 3 lines.
C                   EXCEPT    HDG
C*
C*  When the EXCEPT operation with no entry in factor 2 is
C*  processed, all exception records that do not have an EXCEPT
C*  name specified in positions 30 through 39 are written if the
C*  conditioning indicators are satisfied.  Any exception records
C*  without conditioning indicators and without an EXCEPT name
C*  are always written by an EXCEPT operation with no entry in
C*  factor 2.  In this example,  if indicator 10 is on, TITLE and
C*  AUTH would be printed and then the printer would space 1 line.
C                   EXCEPT
O*
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+.........................
O.............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O          E    10                    1
O                       TITLE
O                       AUTH
O          E            HDG           2
O                       UDATE
O                       PAGE
O          E            HDG           3
O                                             '.............'
O                                             '.............'
O          E            DETAIL        1
O                       AUTH
O                       VERSNO
```

Figure 251. EXCEPT Operations

EXSR (Invoke User Subroutine)

Free-Form Syntax	EXSR *subroutine-name*

Code	Factor 1	Extended Factor 2
EXSR		subroutine-name

The EXSR operation causes the user subroutine named in the *subroutine-name* operand to be processed. The user subroutine name must be a unique symbolic name and must appear as the *subroutine-name* operand of a BEGSR operation. The EXSR operation can appear anywhere in the calculation specifications. When the user subroutine is an exception/error subroutine with a *return-point* operand on it's ENDSR operation, the statement following EXSR is not processed.

The *subroutine-name* operand must be a unique symbolic name or the keyword *TERMSR, *PSSR, or *INZSR:
- *TERMSR specifies that the normal termination subroutine is to be processed
- *PSSR specifies that the program exception/error subroutine is to be processed
- *INZSR specifies the initialization subroutine is to be processed.

You cannot use the EXSR operation to process an action subroutine.

Coding User Subroutines

A user subroutine can be processed from any point in the calculation operations. All operations can be processed within a user subroutine, and in the fixed-format syntax, these operations can be conditioned by any valid indicators in positions 9 through 11. SR or blanks can appear in positions 7 and 8. AND/OR lines within the user subroutine can be indicated in positions 7 and 8.

Fields used in a user subroutine can be defined either in the user subroutine or in another part of the program. In either instance, the fields can be used by both the main program and the user subroutine.

A user subroutine cannot contain another user subroutine. One user subroutine can call another user subroutine; that is, a subroutine can contain an EXSR or CASxx operation. However, an EXSR or CASxx operation within a user subroutine cannot directly call itself. Indirect calls to itself through another subroutine should not be performed because unpredictable results occur. Use the GOTO and TAG operation codes if you want to branch to another point within the same subroutine.

Subroutines do not have to be specified in the order they are used. Each subroutine must have a unique symbolic name and must contain a BEGSR and an ENDSR operation.

The use of the GOTO operation is allowed within a subroutine. GOTO can specify the label on the ENDSR operation associated with that subroutine; it cannot specify the name of a BEGSR operation. A GOTO cannot be issued to a TAG or ENDSR within a subroutine unless the GOTO is in the same subroutine as the TAG or ENDSR. You can use the LEAVESR operation to exit a subroutine from any point within the subroutine. Control passes to the ENDSR operation for the subroutine. Use LEAVESR only from within a subroutine.

See "Coding User Subroutines" on page 577, "Subroutine Operations" on page 378, or "Compare Operations" on page 357 for more information.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*
C                         :
C                         :
C                   EXSR      SUBRTB
C                         :
C                         :
C                   EXSR      SUBRTA
C                         :
C                         :
C     SUBRTA        BEGSR
C                         :
C                         :
C*
C*  One subroutine can call another subroutine.
C*
C                   EXSR      SUBRTC
C                         :
C                         :
C                   ENDSR
C     SUBRTB        BEGSR
C                         :
C                         :
C*
```

Figure 252. Example of Coding User Subroutines - using BEGSR and EXSR

EXTRCT (Extract Date/Time/Timestamp)

Free-Form Syntax	(not allowed - use the %SUBDT built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
EXTRCT (E)		Date/Time: Duration Code	Target	_	ER	_

The EXTRCT operation returns one of the following to the result field:
- The year, month or day part of a date or timestamp field
- The hours, minutes or seconds part of a time or timestamp field
- The microseconds part of the timestamp field

Factor 2 must be a field, subfield, table element, or array element. The Date, Time, or Timestamp followed by the duration code must be specified. For a list of duration codes, see "Date Operations" on page 359.

Factor 1 must be blank.

The result field must be a numeric or character field, a subfield, a table element, or an array element. Character data is left adjusted in the result field.

When using the EXTRCT operation with a Julian Date (format *JUL), specifying a duration code of *D will return the day of the month, specifying *M will return the month of the year. If you require the day and month to be in the 3-digit format, you can use a basing pointer to obtain it.

If a resulting indicator is specified in positions 73 and 74, it is set on when an error occurs during the EXTRCT operation.

To handle EXTRCT exceptions (program status code 112), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "Date Operations" on page 359.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C* Extract the month from a timestamp field to a 2-digit field
C* that is used as an index into a character array containing
C* the names of the months.  Then extract the day from the
C* timestamp to a 2-byte character field which can be used in
C* an EVAL concatenation expression to form a string containing
C* for example "March 13"
C
C                   EXTRCT    LOGONTIME:*M  LOGMONTH          2 0
C                   EXTRCT    LOGONTIME:*D  LOGDAY            2
C                   EVAL      DATE_STR = %TRIMR(MONTHS(LOGMONTH)
C                                        + ' ' + LOGDAY
```

Figure 253. EXTRCT Operations

FEOD (Force End of Data)

Free-Form Syntax	FEOD{(EN)} *file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
FEOD (EN)		file-name		_	ER	_

The FEOD operation signals the logical end of data for an OS/400 file. The file can be used again for subsequent file operations without specifying the OPEN operation. The file is still connected to the program. This is different than the CLOSE operation where the file is disconnected from the program and you must specify an OPEN if you wish to use the file again. For more information, see "CLOSE (Close Files)" on page 542.

The FEOD operation can only be used with OS/400 files.

The *file-name* operand names the file to which FEOD is specified.

Operation extender N may be specified for an FEOD to an output-capable DISK file that uses blocking. (see "Blocking Considerations" on page 47) If operation extender N is specified, any unwritten records in the block will be written out to the database, but they will not necessarily be written to non-volatile storage. Using the N extender can improve performance.

To handle FEOD exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

To process any further sequential operations after the FEOD operation (for example, using READ or READP), you must reposition the file.

The FEOD operation flushes any buffered data for output files. The data is written to DISK or PRINTER.

For more information, see "File Operations" on page 363.

FOR (For)

Free-Form Syntax	FOR{(MR)} *index-name* {= *start-value*} {BY *increment*} {TO|DOWNTO *limit*}

Code	Factor 1	Extended Factor 2
FOR		index-name = start-value BY increment TO | DOWNTO limit

*** Restriction:** The FOR operation is unsupported in Java applications.

The FOR operation begins a group of operations and controls the number of times the group will be processed. To indicate the number of times the group of operations is to be processed, specify an index name, a starting value, an increment value, and a limit value. The optional starting, increment, and limit values can be a free-form expressions. An associated END or ENDFOR statement marks the end of the group. For further information on FOR groups, see "Structured Programming Operations" on page 376.

The syntax of the FOR operation is as follows:

```
FOR         index-name { = starting-value }
            { BY increment-value }
            { TO | DOWNTO limit-value }
   { loop body }
ENDFOR | END
```

The starting-value, increment-value, and limit-value can be numeric values or expressions with zero decimal positions. The increment value, if specified, cannot be zero.

The BY and TO (or DOWNTO) clauses can be specified in either order. Both ″BY 2 TO 10″ and ″TO 10 BY 2″ are allowed.

In addition to the FOR operation itself, the conditioning indicators on the FOR and ENDFOR (or END) statements control the FOR group. The conditioning indicators on the FOR statement control whether or not the FOR operation begins. These indicators are checked only once, at the beginning of the for loop. The conditioning indicators on the associated END or ENDFOR statement control whether or not the FOR group is repeated another time. These indicators are checked at the end of each loop.

The FOR operation is performed as follows:

1. If the conditioning indicators on the FOR statement line are satisfied, the FOR operation is processed (step 2). If the indicators are not satisfied, control passes to the next operation to be processed following the associated END or ENDFOR statement (step 8).
2. If specified, the initial value is assigned to the index name. Otherwise, the index name retains the same value it had before the start of the loop.
3. If specified, the limit value is evaluated and compared to the index name. If no limit value is specified, the loop repeats indefinitely until it encounters a statement that exits the loop (such as a LEAVE or GOTO) or that ends the program or procedure (such as a RETURN).

   If the TO clause is specified and the index name value is greater than the limit value, control passes to the first statement following the ENDFOR statement. If DOWNTO is specified and the index name is less than the limit value, control passes to the first statement after the ENDFOR.

4. The operations in the FOR group are processed.
5. If the conditioning indicators on the END or ENDFOR statement are not satisfied, control passes to the statement after the associated END or ENDFOR and the loop ends.
6. If the increment value is specified, it is evaluated. Otherwise, it defaults to 1.
7. The increment value is either added to (for TO) or subtracted from (for DOWNTO) the index name. Control passes to step 3. (Note that the conditioning indicators on the FOR statement are not tested again (step 1) when control passes to step 3.)
8. The statement after the END or ENDFOR statement is processed when the conditioning indicators on the FOR, END, or ENDFOR statements are not satisfied (step 1 or 5), or when the index value is greater than (for TO) or less than (for DOWNTO) the limit value (step 3), or when the index value overflows.

Note: If the FOR loop is performed *n* times, the limit value is evaluated *n+1* times and the increment value is evaluated *n* times. This can be important if the limit value or increment value is complex and time-consuming to evaluate, or if the limit value or increment value contains calls to subprocedures with side-effects. If multiple evaluation of the limit or increment is not desired, calculate the values in temporaries before the FOR loop and use the temporaries in the FOR loop.

Remember the following when specifying the FOR operation:
- The index name cannot be declared on the FOR operation. Variables should be declared in the Definition specifications.
- The *index-name* can be any fully-qualified name, including an indexed array element.

See "LEAVE (Leave a Do/For Group)" on page 596 and "ITER (Iterate)" on page 591 for information on how those operations affect a FOR operation.

For more information, see "Structured Programming Operations" on page 376.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/free
   // Example 1
   // Compute n!

   factorial = 1;
   for i = 1 to n;
      factorial = factorial * i;
   endfor;

   // Example 2
   // Search for the last nonblank character in a field.
   // If the field is all blanks, "i" will be zero.
   // Otherwise, "i" will be the position of nonblank.

   for i = %len (field) downto 1;
      if %subst(field: i: 1) <> ' ';
         leave;
      endif;
   endfor;

   // Example 3
   // Extract all blank-delimited words from a sentence.

   WordCnt = 0;
   for i = 1 by WordIncr to %len (Sentence);
      // Is there a blank?
      if %subst(Sentence: i: 1) = ' ';
         WordIncr = 1;
         iter;
      endif;

      // We've found a word - determine its length:
      for j = i+1 to %len(Sentence);
         if %subst (Sentence: j: 1) = ' ';
            leave;
         endif;
      endfor;

      // Store the word:
      WordIncr = j - i;
      WordCnt = WordCnt + 1;
      Word (WordCnt) = %subst (Sentence: i: WordIncr);
   endfor;

/end-free
```

Figure 254. Examples of the FOR Operation

GETATR (Retrieve Attribute)

Free-Form Syntax	(not allowed - use the %GETATR built-in function or “Qualified GUI Part Attribute Access” on page 379)

Code	Factor 1	Factor 2	Result Field	Indicators		
GETATR (E)	part name	attribute	field name	_	ER	_

The GETATR operation retrieves the value of a part’s attribute. The parent window name is the default window name. A part’s attribute can be retrieved only if that part has been created.

Notes:

1. The GETATR operations can be used for multiple link action subroutines. For a description of multiple link action subroutines, see “BEGACT (Begin Action Subroutine)” on page 508. To retrieve an attribute for a part on a window other than the parent window, use the %GETATR built-in function. For a description of the %GETATR built-in function, see “%GETATR (Retrieve Attribute)” on page 446.
2. The GETATR operation does not support 1-byte and 8-byte signed and unsigned integer values, and unicode values.

Factor 1 must contain the name of a part (which must be a character literal) or a field name that contains the name of a part (which must be characters).

Factor 2 must contain the name of the attribute being retrieved (which must be a character literal) or a field name that contains the name of an attribute (which must be characters).

Factor 1 and factor 2 cannot contain graphic characters.

The result field must contain the name of the field which contains the retrieved value of the attribute. The type of the result field must be the same as the attribute type.

To handle GETATR exceptions, either the operation code extender ’E’ or an error indicator ER can be specified, but not both. For more information on error handling, see “Program Exception and Errors” on page 51.

Note: The GETATR operation does not affect the corresponding program fields for parts. If you want the corresponding program field for the part to contain the current value of an entry field, make it the target of the operation.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C* Retrieve the part type on a part called MLE01.
C*
C     'MLE01'       GETATR    'PartType'    Mle
```

Figure 255. GETATR Operation

GOTO (Go To)

Free-Form Syntax	(not allowed - use other operation codes, such as LEAVE, LEAVESR, ITER, and RETURN)

Code	Factor 1	Factor 2	Result Field	Indicators		
GOTO		Label				

The GOTO operation allows calculation operations to be skipped by instructing the program to go to (or branch to) a specified label in the program. A TAG operation names the destination of a GOTO operation. The TAG can either precede or follow the GOTO.

A GOTO within a subroutine in the main procedure can be issued to a TAG within the same subroutine. A GOTO within a subroutine in a subprocedure can be issued to a TAG within the same subroutine, or within the body of the subprocedure.

Factor 2 must contain the label to which the program is to branch. This label is entered in factor 1 of a TAG or ENDSR operation. The label must be a unique symbolic name.

For a description of the TAG operation, see “TAG (Tag)” on page 699.

For more information, see “Branching Operations” on page 352.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  If indicator 10, 15, or 20 is on, the program branches to
C*  the TAG label specified in the GOTO operations.
C*
C   10              GOTO      RTN1
C*
C*
C   15              GOTO      RTN2
C*
C     RTN1          TAG
C*
C                     :
C                     :
C   20              GOTO      END
C*
C                     :
C                     :
C     END           TAG
```

Figure 256. GOTO and TAG Operations

IF (If)

Free-Form Syntax	IF{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2
IF (M/R)	Blank	indicator-expression

The IF operation allows a series of operation codes to be processed if a condition is met. Its function is similar to that of the IFxx operation code. It differs in that the logical condition is expressed by an indicator valued expression (*indicator-expression*). The operations controlled by the IF operation are performed when the expression in the *indicator-expression* is true.

For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 390. "Compare Operations" on page 357 describes the rules for specifying the compare operations.

For more information, see "Structured Programming Operations" on page 376.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++..
C                            Extended-factor2-continuation+++++++++++++++++
C* The operations controlled by the IF operation are performed
C* when the expression is true.  That is A is greater than 10 and
C* indicator 20 is on.
C
C                   IF      A>10 AND *IN(20)
C                   :
C                   ENDIF
C*
C* The operations controlled by the IF operation are performed
C* when Date1 represents a later date then Date2
C
C                   IF      Date1 > Date2
C                   :
C                   ENDIF
C*
```

Figure 257. IF Operations

IFxx (If)

Free-Form Syntax	(not allowed - use the IF operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
IFxx	Comparand	Comparand				

The IFxx operation allows a group of calculations to be processed if a certain relationship, specified by xx, exists between factor 1 and factor 2. When ANDxx and ORxx operations are used with IFxx, the group of calculations is performed if the condition specified by the combined operations exists. (For the meaning of xx, see "Structured Programming Operations" on page 376.)

Factor 1 and factor 2 must contain a literal, a named constant, a figurative constant, a table name, an array element, a data structure name, or a field name. Both the factor 1 and factor 2 entries must be of the same data type.

If the relationship specified by the IFxx and any associated ANDxx or ORxx operations does not exist, control passes to the calculation operation immediately following the associated ENDIF operation. If an ELSE operation is specified as well, control passes to the first calculation operation that can be processed following the ELSE operation.

Conditioning indicator entries on the ENDIF operation associated with IFxx must be blank.

An ENDIF statement must be used to close an IFxx group. If an IFxx statement is followed by an ELSE statement, an ENDIF statement is required after the ELSE statement but not after the IFxx statement.

You have the option of indenting DO statements, IF-ELSE clauses, and SELECT-WHENxx-OTHER clauses in the compiler listing for readability. See the online help for the Project>Build Options dialog in the GUI Designer for a description of the compiler options.

"Compare Operations" on page 357 describes the rules for specifying the compare operations.

For more information, see "Compare Operations" on page 357 or "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  If FLDA equals FLDB, the calculation after the IFEQ operation
C*  is processed.  If FLDA does not equal FLDB, the program
C*  branches to the operation immediately following the ENDIF.
C
C     FLDA          IFEQ      FLDB
C                       :
C                       :
C                   ENDIF
C
C*  If FLDA equals FLDB, the calculation after the IFEQ operation
C*  is processed and control passes to the operation immediately
C*  following the ENDIF statement.  If FLDA does not equal FLDB,
C*  control passes to the ELSE statement and the calculation
C*  immediately following is processed.
C
C     FLDA          IFEQ      FLDB
C                       :
C                       :
C                   ELSE
C                       :
C                       :
C                   ENDIF
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  If FLDA is equal to FLDB and greater than FLDC, or, if FLDD
C*  is equal to FLDE and greater than FLDF, the calculation
C*  after the ANDGT operation is processed.  If neither of the
C*  specified conditions exists, the program branches to the
C*  operation immediately following the ENDIF statement.
C
C     FLDA          IFEQ      FLDB
C     FLDA          ANDGT     FLDC
C     FLDD          OREQ      FLDE
C     FLDD          ANDGT     FLDF
C                       :
C                       :
C                   ENDIF
```

Figure 258. IFxx/ENDIF and IFxx/ELSE/ENDIF Operations

IN (Retrieve a Data Area)

Free-Form Syntax	IN{(E)} {*LOCK} *data-area-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
IN (E)	*LOCK	data-area-name		_	ER	_

The IN operation retrieves a data area.

The reserved word *LOCK can be specified in Factor 1 to indicate that the data area cannot be updated or locked by another program until (1) an UNLOCK operation is processed, (2) an OUT operation with no *data-area-name* operand specified, or (3) the program implicitly unlocks the data area when the program ends

If a data area is locked, it can be read but not updated by other programs.

data-area-name must be the name of a definition defined with the DTAARA keyword, the result field of a *DTAARA DEFINE operation, or the reserved word *DTAARA. When *DTAARA is specified, all data areas defined in the program are retrieved.

If name of the data area is determined at runtime because DTAARA(*VAR) was specified on the definition of the field, then the variable containing the name of the data area must be set before the IN operation. However, if the data area is already locked due to a prior *LOCK IN operation, the variable containing the name will not be consulted; instead, the previously locked data area will be used.

To handle IN exceptions (program status codes 401-421, 431, or 432), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

On a fixed-form calculation, positions 71-72 and 75-76 must be blank.

For a description of general rules, see "Data-Area Operations" on page 358.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  TOTAMT, TOTGRS, and TOTNET are defined as data areas.  The IN
C*  operation retrieves all the data areas defined in the program
C*  and locks them.  The program processes calculations, and then
C*  writes and unlocks all the data areas.
C*  The data areas can then be used by other programs.
C*
C     *LOCK         IN        *DTAARA
C                   ADD       AMOUNT        TOTAMT
C                   ADD       GROSS         TOTGRS
C                   ADD       NET           TOTNET
C
C                   OUT       *DTAARA
C
C*
C* Define Data areas
C*
C     *DTAARA       DEFINE                  TOTAMT            8 2
C     *DTAARA       DEFINE                  TOTGRS           10 2
C     *DTAARA       DEFINE                  TOTNET           10 2
```

Figure 259. IN and OUT Operations

ITER (Iterate)

Free-Form Syntax	ITER

Code	Factor 1	Factor 2	Result Field	Indicators		
ITER						

The ITER operation transfers control from within a DO or FOR group to the ENDDO or ENDFOR statement of the do group. It can be used in DO, DOU, DOUxx, DOW, DOWxx, and FOR loops to transfer control immediately to a loop's ENDDO or ENDFOR statement. It causes the next iteration of the loop to be executed immediately. ITER affects the innermost loop.

If conditioning indicators are specified on the ENDDO or ENDFOR statement to which control is passed, and the condition is not satisfied, processing continues with the statement following the ENDDO or ENDFOR operation.

The LEAVE operation is similar to the ITER operation; however, LEAVE transfers control to the statement ***following*** the ENDDO or ENDFOR operation.

For more information, see "Branching Operations" on page 352 or "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following example uses a DOU loop containing a DOW loop.
C* The IF statement checks indicator 01.  If indicator 01 is ON,
C* the LEAVE operation is executed, transferring control out of
C* the innermost DOW loop to the Z-ADD instruction.  If indicator
C* 01 is not ON, subroutine PROC1 is processed.  Then indicator
C* 12 is checked.  If it is OFF, ITER transfers control to the
C* innermost ENDDO and the condition on the DOW is evaluated
C* again.  If indicator 12 is ON, subroutine PROC2 is processed.
C
C                   DOU       FLDA = FLDB
C                   :
C     NUM           DOWLT     10
C                   IF        *IN01
C                   LEAVE
C                   ENDIF
C                   EXSR      PROC1
C     *IN12         IFEQ      *OFF
C                   ITER
C                   ENDIF
C                   EXSR      PROC2
C                   ENDDO
C                   Z-ADD     20            RSLT              2 0
C                   :
C                   ENDDO
C                   :
```

Figure 260. ITER Operation (Part 1 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following example uses a DOU loop containing a DOW loop.
C* The IF statement checks indicator 1. If indicator 01 is ON, the
C* MOVE operation is executed, followed by the LEAVE operation,
C* transferring control from the innermost DOW loop to the Z-ADD
C* instruction.  If indicator 01 is not ON, ITER transfers control
C* to the innermost ENDDO and the condition on the DOW is
C* evaluated again.
C                   :
C     FLDA          DOUEQ     FLDB
C                   :
C     NUM           DOWLT     10
C     *IN01         IFEQ      *ON
C                   MOVE      'UPDATE'      FIELD           20
C                   LEAVE
C                   ELSE
C                   ITER
C                   ENDIF
C                   ENDDO
C                   Z-ADD     20            RSLT             2 0
C                   :
C                   ENDDO
C                   :
```

Figure 260. ITER Operation (Part 2 of 2)

KFLD (Define Parts of a Key)

Free-Form Syntax	(not allowed - use %KDS)

Code	Factor 1	Factor 2	Result Field	Indicators		
KFLD			Key field			

The KFLD operation indicates that a field is part of a search argument identified by a KLIST name.

The KFLD operation can be specified anywhere within calculations, but must follow a KLIST or KFLD operation. Conditioning indicator entries (positions 9 through 11) are not permitted.

KFLDs can be global or local. A KLIST in a main procedure can have only global KFLDs associated with it. A KLIST in a subprocedure can have local and global KFLDs.

Factor 2 can contain an indicator for a null-capable key field if the **User control** option or **ALWNULL(*USRCTL)** keyword is specified.

If the indicator is on, the key fields with null values are selected. If the indicator is off or not specified, the key fields with null values are not selected. See "Keyed Operations" on page 141 for information on how to access null-capable keys.

The result field must contain the name of a field that is to be part of the search argument. The result field cannot contain an array name. Each KFLD field must agree in length, data type, and decimal position with the corresponding field in the composite key of the record or file. However, if the record has a variable-length KFLD field, the corresponding field in the composite key must be varying but does not need to be the same length. Each KFLD field need not have the same name as the corresponding field in the composite key. The order the KFLD fields are specified in the KLIST determines which KFLD is associated with a particular field in the composite key. For example, the first KFLD field following a KLIST operation is associated with the leftmost (high-order) field of the composite key.

Graphic and UCS-2 key fields must have the same CCSID as the key in the file.

Figure 261 on page 595 shows an example of the KLIST operation with KFLD operations.

For more information, see "Declarative Operations" on page 362.

KLIST (Define a Composite Key)

Free-Form Syntax	(not allowed - use %KDS)

Code	Factor 1	Factor 2	Result Field	Indicators		
KLIST	KLIST name					

The KLIST operation gives a name to a list of KFLDs. This list is used as a search argument to retrieve records from externally described files that have a composite key. A composite key is a key that contains a list of key fields. It is built from left to right. The first KFLD specified is the leftmost (high-order) field of a composite key.

A KLIST must be followed immediately by at least one KFLD. A KLIST is ended when a non-KFLD operation is encountered. If a search argument is composed of more than one field (a composite key), you must specify a KLIST with multiple KFLDs. The same KLIST name can be used as the search argument for multiple files, or it can be used multiple times as the search argument for the same file.

Factor 1 must contain a unique name. This name can appear in factor 1 of a CHAIN, DELETE, READE, READPE, SETGT, or SETLL operation.

Conditioning indicator entries (positions 9 through 11) are not permitted.

A KLIST in a main procedure can have only local KFLDs associated with it. A KLIST in a subprocedure can have local and global KFLDs.

For more information, see “Declarative Operations” on page 362.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
A*  DDS source
A          R RECORD
A            FLDA           4
A            SHIFT          1  0
A            FLDB          10
A            CLOCK#         5  0
A            FLDC          10
A            DEPT           4
A            FLDD           8
A          K DEPT
A          K SHIFT
A          K CLOCK#
A*
A*  End of DDS source
A*
A******************************************************************
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The KLIST operation indicates the name, FILEKY, by which the
C*  search argument can be specified.
C     FILEKY        KLIST
C                   KFLD                    DEPT
C                   KFLD                    SHIFT
C                   KFLD                    CLOCK#
```

Figure 261. KLIST and KFLD Operations

LEAVE (Leave a Do/For Group)

Free-Form Syntax	LEAVE

Code	Factor 1	Factor 2	Result Field	Indicators		
LEAVE						

The LEAVE operation transfers control from within a DO or FOR group to the statement following the ENDDO or ENDFOR operation.

You can use LEAVE within a DO, DOU, DOUxx, DOW, DOWxx, or FOR loop to transfer control immediately from the innermost loop to the statement following the innermost loop's ENDDO or ENDFOR operation. Using LEAVE to leave a DO or FOR group does not increment the index.

In nested loops, LEAVE causes control to transfer outwards by one level only. LEAVE is not allowed outside a DO or FOR group.

The ITER operation is similar to the LEAVE operation; however, ITER transfers control ***to*** the ENDDO or ENDFOR statement.

For more information, see "Branching Operations" on page 352 or "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following example uses an infinite loop.  When the user
C* types 'q', control transfers to the LEAVE operation, which in
C* turn transfers control out of the loop to the Z-ADD operation.
C*
C     2             DOWNE     1
C                   :
C                   IF        ANSWER = 'q'
C                   LEAVE
C                   ENDIF
C                   :
C                   ENDDO
C                   Z-ADD     A             B
C*
C* The following example uses a DOUxx loop containing a DOWxx.
C* The IF statement checks indicator 1. If it is ON, indicator
C* 99 is turned ON, control passes to the LEAVE operation and
C* out of the inner DOWxx loop.
C*
C* A second LEAVE instruction is then executed because indicator 99
C* is ON, which in turn transfers control out of the DOUxx loop.
C*
C                   :
C     FLDA          DOUEQ     FLDB
C     NUM           DOWLT     10
C     *IN01         IFEQ      *ON
C                   SETON                                        99
C                   LEAVE
C                   :
C                   ENDIF
C                   ENDDO
C  99               LEAVE
C                   :
C                   ENDDO
C                   :
```

Figure 262. LEAVE Operation

LEAVESR (Leave a Subroutine)

Free-Form Syntax	LEAVESR

Code	Factor 1	Factor 2	Result Field	Indicators		
LEAVESR						

The LEAVESR operation exits a subroutine from any point within the subroutine. Control passes to the ENDSR operation for the subroutine. LEAVESR is allowed only from within a subroutine.

The control level entry (positions 7 and 8) can be SR or blank. Conditioning indicator entries (positions 9 to 11) can be specified.

For more information, see "Subroutine Operations" on page 378.

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
 *
C     CheckCustName BEGSR
C     Name          CHAIN     CustFile
 *
 * Check if the name identifies a valid customer
 *
C                   IF        not %found(CustFile)
C                   EVAL      Result = CustNotFound
C                   LEAVESR
C                   ENDIF
 *
 * Check if the customer qualifies for discount program
C                   IF        Qualified = *OFF
C                   EVAL      Result = CustNotQualified
C                   LEAVESR
C                   ENDIF
 *
 * If we get here, customer can use the discount program
C                   EVAL      Result = CustOK
C                   ENDSR
```

Figure 263. LEAVESR Operations

LOOKUP (Look Up a Table or Array Element)

Free-Form Syntax	(not allowed - use the %LOOKUP or %TLOOKUP built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
LOOKUP						
(array)	Search argument	Array name		HI	LO	EQ
(table)	Search argument	Table name	Table name	HI	LO	EQ

The LOOKUP operation searches an array or table for an element. The search argument and the table or array must have the same type and length (except Time and Date fields which can have a different length). If the array or table is fixed-length character, graphic, or UCS-2, the search argument must also be fixed-length. For variable length, the length of the search argument can have a different length from the array or table. A sequence for the table or array must be specified on the definition specification using the ASCEND or DESCEND keywords.

Factor 1 must be a literal, a field name, an array element, a table name, a named constant, or a figurative constant. The nature of the comparison depends on the data type:

Graphic and UCS-2 data
The comparison is hexadecimal.

Numeric data
Decimal alignment is not processed.

Other data types
The considerations for comparison described in "Compare Operations" on page 357 apply to other types.

For a table LOOKUP, the search argument is the element of the table last selected in a LOOKUP operation. If the last LOOKUP operation has not been processed, the first element of the table is used as the search argument. If the result field is specified for a table LOOKUP, it must contain the name of a second table. The position of the elements in the second table correspond to the position of the elements in the first table. The LOOKUP operation retrieves the element from the second table.

For an array LOOKUP, an index can be used. The LOOKUP begins with the element specified by the index. The index value is set to the position number of the element located. If the index is equal to zero or is higher than the number of elements in the array when the search begins, an error occurs. The index is set to 1 if the search is unsuccessful. If the index is a named constant, the index value does not change.

Resulting indicators must be specified to determine the search to be done and then to reflect the result of the search. A sequence for the table or array must also be specified on the definition specification using the ASCEND or DESCEND keywords. Any specified indicator is set on only if the search is successful. No more than two indicators can be used. Resulting indicators can be assigned to equal and high or to equal and low. The program searches for an entry that

satisfies either condition with equal given precedence; that is, if no equal entry is found, the nearest lower or nearest higher entry is selected.

If an indicator is specified in positions 75-76, the %EQUAL built-in function returns '1' if an element is found that exactly matches the search argument. The %FOUND built-in function returns '1' if any specified search is successful.

Resulting indicators can be assigned to equal and low, or equal and high. High and low cannot be specified on the same LOOKUP operation. The compiler assumes a sorted, sequenced array or table when a high or low indicator is specified for the LOOKUP operation. The LOOKUP operation searches for an entry that satisfies the low/equal or high/equal condition with equal given priority.

- *High (71-72):* Instructs the program to find the entry that is nearest to, yet higher in sequence than, the search argument. The first higher entry found sets the indicator assigned to high on.
- *Low (73-74):* Instructs the program to find the entry that is nearest to, yet lower in sequence than, the search argument. The first such entry found sets the indicator assigned to low on.
- *Equal (75-76):* Instructs the program to find the entry equal to the search argument. The first equal entry found sets the indicator assigned to equal on.

If the equal indicator is the only indicator specified, the entire array or table is searched. If the table or array is in ascending sequence, and you want an equal comparison, specify the High indicator. An entire search of the table or array does not occur.

For more information, see "Array Operations" on page 351.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  In this example, the programmer wants to know which element in
C*  ARY the LOOKUP operation locates.  The Z-ADD operation sets the
C*  field X to 1.  The LOOKUP starts at the element ARY that is
C*  indicated by field X and continues running until it finds the
C*  first element equal to SRCHWD.  The index value, X, is set to
C*  the position number of the element located.
C
C                   Z-ADD     1             X               3 0
C     SRCHWD        LOOKUP    ARY(X)                                26
C
C*  In this example, the programmer wants to know if an element
C*  is found that is equal to SRCHWD.  LOOKUP searches ARY until it
C*  finds the first element equal to SRCHWD.  When this occurs,
C*  indicator 26 is set on and %EQUAL is set to return '1'.
C
C     SRCHWD        LOOKUP    ARY                                   26
C
C*  The LOOKUP starts at a variable index number specified by
C*  field X.  Field X does not have to be set to 1 before the
C*  LOOKUP operation.  When LOOKUP locates the first element in
C*  ARY equal to SRCHWD, indicator 26 is set on and %EQUAL is set
C*  to return '1'.  The index value, X, is set to the position
C*  number of the element located.
C*
C
C     SRCHWD        LOOKUP    ARY(X)                                26
```

Figure 264. LOOKUP Operation with Arrays

```
 * In this example, an array of customer information actually consists
 * of several subarrays.  You can search either the main array or the
 * subarrays overlaying the main array.
D custInfo        DS
D  cust                               DIM(100)
D   name                        30A   OVERLAY(cust : *NEXT)
D   id_number                   10I 0 OVERLAY(cust : *NEXT)
D   amount                      15P 3 OVERLAY(cust : *NEXT)

 * You can search for a particular set of customer information
 * by doing a search on the "cust" array
C     custData      LOOKUP    cust(i)                                 10

 * You can search on a particular field of the customer information
 * by doing a search on one of the overlay arrays
C     custName      LOOKUP    name(i)                                 11
 * After the search, the array index can be used with any of the
 * overlaying arrays.  If the search on name(i) is successful,
 * the id_number and amount for that customer are available
 * in id_number(i) and amount(i).
```

Figure 265. LOOKUP Operation with Subarrays

MONITOR (Begin a Monitor Group)

Free-Form Syntax	MONITOR

Code	Factor 1	Factor 2	Result Field	Indicators		
MONITOR						

The monitor group performs conditional error handling based on the status code. It consists of:
- A MONITOR statement
- One or more ON-ERROR groups
- An ENDMON statement.

After the MONITOR statement, control passes to the next statement. The monitor block consists of all the statements from the MONITOR statement to the first ON-ERROR statement. If an error occurs when the monitor block is processed, control is passed to the appropriate ON-ERROR group.

If all the statements in the MONITOR block are processed without errors, control passes to the statement following the ENDMON statement.

The monitor group can be specified anywhere in calculations. It can be nested within IF, DO, SELECT, or other monitor groups. The IF, DO, and SELECT groups can be nested within monitor groups.

If a monitor group is nested within another monitor group, the innermost group is considered first when an error occurs. If that monitor group does not handle the error condition, the next group is considered.

Conditioning indicators can be used on the MONITOR statement. If they are not satisfied, control passes immediately to the statement following the ENDMON statement of the monitor group. Conditioning indicators cannot be used on ON-ERROR operations individually.

If a monitor block contains a call to a subprocedure, and the subprocedure has an error, the subprocedure's error handling will take precedence. For example, if the subprocedure has a *PSSR subroutine, it will get called. The MONITOR group containing the call will only be considered if the subprocedure fails to handle the error and the call fails with the error-in-call status of 00202.

The monitor group does handle errors that occur in a subroutine. If the subroutine contains its own monitor groups, they are considered first.

Branching operations are not allowed within a monitor block, but are allowed within an ON-ERROR block.

A LEAVE or ITER operation within a monitor block applies to any active DO group that contains the monitor block. A LEAVESR or RETURN operation within a monitor block applies to any subroutine, subprocedure, or procedure that contains the monitor block.

For more information, see "Error-Handling Operations" on page 362.

```
 * The MONITOR block consists of the READ statement and the IF
 * group.
 * - The first ON-ERROR block handles status 1211 which
 *   is issued for the READ operation if the file is not open.
 * - The second ON-ERROR block handles all other file errors.
 * - The third ON-ERROR block handles the string-operation status
 *   code 00100 and array index status code 00121.
 * - The fourth ON-ERROR block (which could have had a factor 2
 *   of *ALL) handles errors not handled by the specific ON-ERROR
 *   operations.
 *
 * If no error occurs in the MONITOR block, control passes from the
 * ENDIF to the ENDMON.
C                   MONITOR
C                   READ       FILE1
C                   IF         NOT %EOF
C                   EVAL       Line = %SUBST(Line(i) :
C                                            %SCAN('***': Line(i)) + 1)
C                   ENDIF
C                   ON-ERROR   1211
C                    ... handle file-not-open
C                   ON-ERROR   *FILE
C                    ... handle other file errors
C                   ON-ERROR   00100 : 00121
C                    ... handle string error and array-index error
C                   ON-ERROR
C                    ... handle all other errors
C                   ENDMON
```

Figure 266. MONITOR Operation

MOVE (Move)

Free-Form Syntax	(not allowed - use the EVAL or EVALR operations, or built-in functions such as %CHAR, %DATE, %DEC , %DECH, %GRAPH, %INT, %INTH, %TIME, %TIMESTAMP , %UCS2, %UNS, or %UNSH)

Code	Factor 1	Factor 2	Result Field	Indicators		
MOVE (P)	Data Attributes	Source field	Target field	+	–	ZB

The MOVE operation transfers characters from factor 2 to the result field. Moving starts with the rightmost character of factor 2.

When moving Date, Time, or Timestamp fields, factor 1 must be blank unless either the source or the target is a character or numeric field.

Otherwise, factor 1 contains the date or time format compatible with the character or numeric field that is the source of the operation. For information on the formats that can be used see “Date Data” on page 119, “Time Data” on page 135, and “Timestamp Data” on page 137.

If the source or target is a character field, you may optionally indicate the separator following the format in factor 1. Only separators that are valid for that format are allowed.

If factor 2 is *DATE or UDATE and the result is a Date field, factor 1 is not required. If factor 1 contains a date format it must be compatible with the format of *DATE or UDATE as specified by the DATEDIT keyword on the control specification.

When moving character, graphic, UCS-2, or numeric date, if factor 2 is longer than the result field, the excess leftmost characters or digits of factor 2 are not moved. If the result field is longer than factor 2, the excess leftmost characters or digits in the result field are unchanged, unless padding is specified.

You cannot specify resulting indicators if the result field is an array; you can specify them if it is an array element, or a non-array field.

If factor 2 is shorter than the length of the result field, a P specified in the operation extender position causes the result field to be padded on the left after the move occurs.

Float numeric fields and literals are not allowed as Factor 2 or Result-Field entries.

If CCSID(*GRAPH : IGNORE) is specified or assumed for the module, MOVE operations between UCS-2 and graphic data are not allowed.

When moving variable-length character, graphic, or UCS-2 data, the variable-length field works in exactly the same way as a fixed-length field with the same current length. For examples, see Figures 273 to 278.

The tables which appear following the examples (see “MOVE Examples (Part 1)” on page 605), show how data is moved from factor 2 to the result field. For further information on the MOVE operation, see “Move Operations” on page 368 or “Conversion Operations” on page 358.

MOVE Examples (Part 1)

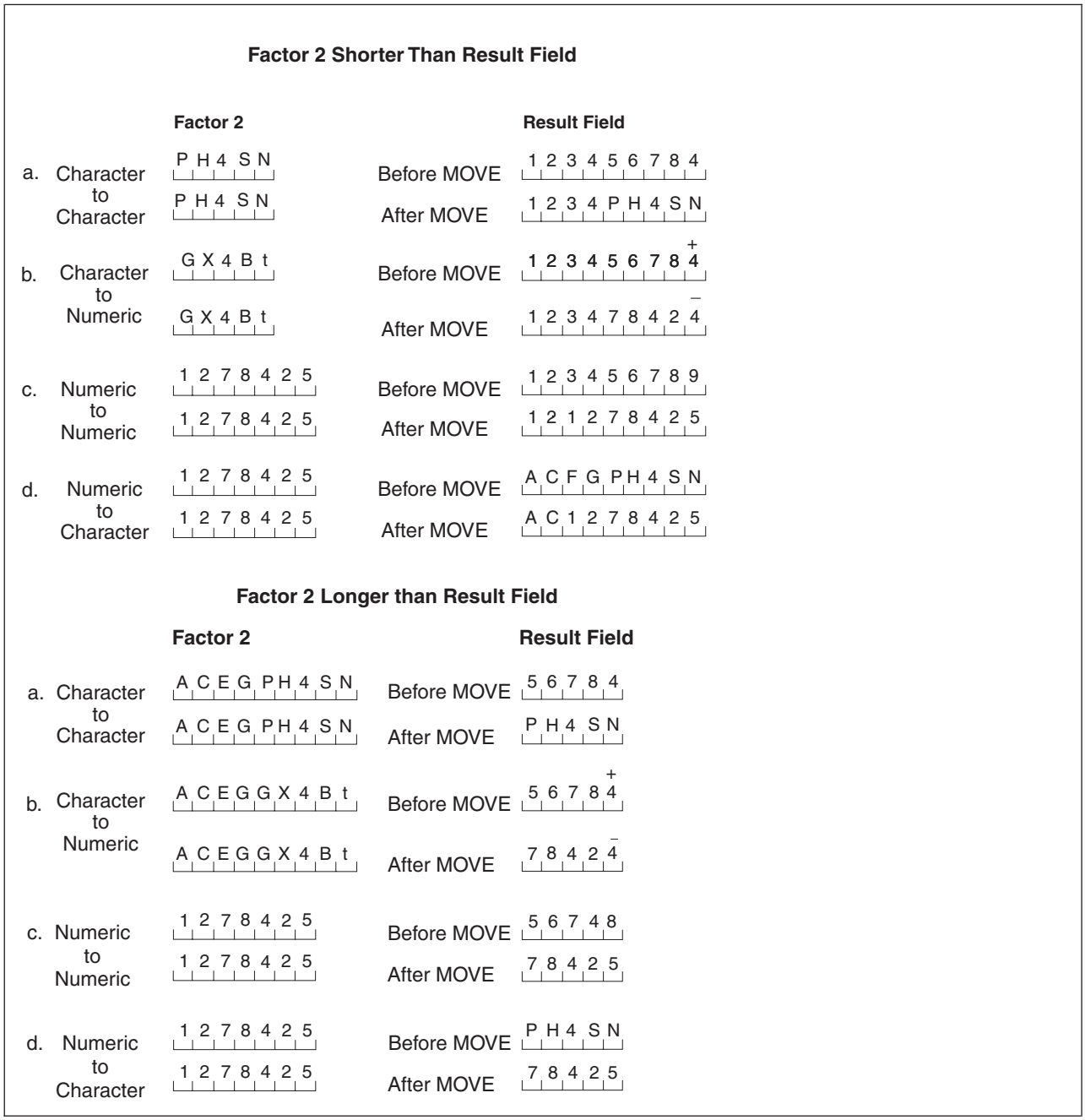
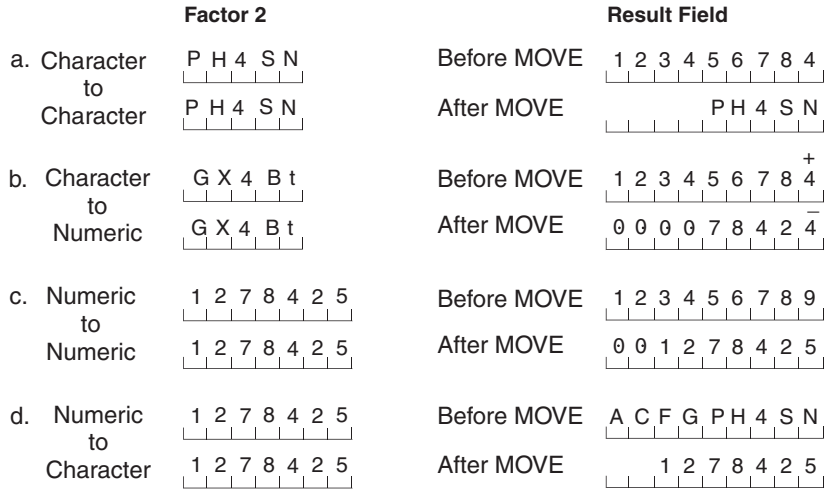
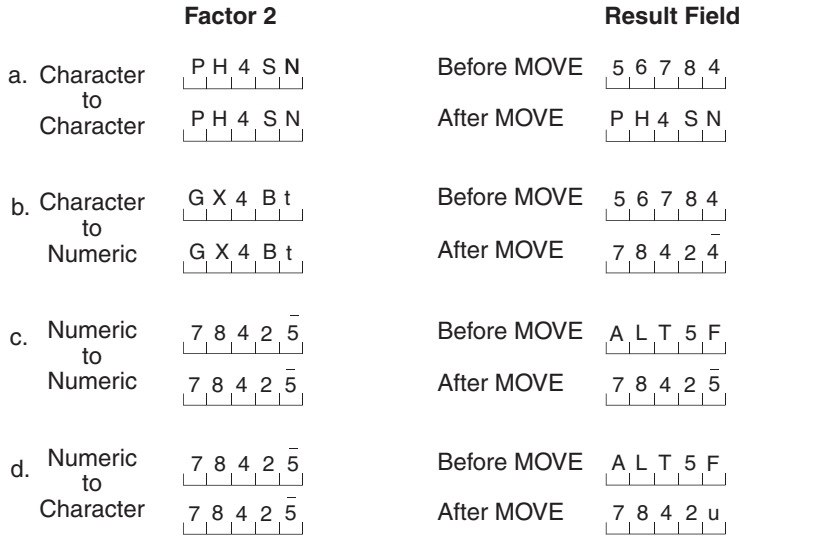


Figure 267. MOVE Operation (Part 1 of 2)

Factor 2 Shorter Than Result Field With P in Operation Extender Field



Factor 2 and Result Field Same Length



Note: $\bar{4}$ = letter t , and $\bar{5}$ = letter u.

Figure 267. MOVE Operation (Part 2 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
H*  Control specification date format
H*
H DATFMT(*ISO)
H
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++
D*
D DATE_ISO       S               D
D DATE_YMD       S               D   DATFMT(*YMD)
D                                    INZ(D'1992-03-24')
D DATE_EUR       S               D   DATFMT(*EUR)
D                                    INZ(D'2197-08-26')
D DATE_JIS       S               D   DATFMT(*JIS)
D NUM_DATE1      S              6P 0 INZ(210991)
D NUM_DATE2      S              7P 0
D CHAR_DATE      S              8    INZ('02/01/53')
D CHAR_LONGJUL   S              8A   INZ('2039/166')
D DATE_USA       S               D   DATFMT(*USA)
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+H1LoEq..
C*
C* Move between Date fields. DATE_EUR will contain 24.03.1992
C*
C                   MOVE      DATE_YMD      DATE_EUR
C*
C* Convert numeric value in ddmmyy format into a *ISO Date.
C* DATE_ISO will contain 1991-09-21 after each of the 2 moves.
C*
C     *DMY          MOVE      210991        DATE_ISO
C     *DMY          MOVE      NUM_DATE      DATE_ISO
C*
C* Move a character value representing a *MDY date to a *JIS Date.
C* DATE_JIS will contain 1953-02-01 after each of the 2 moves.
C*
C     *MDY/         MOVE      '02/01/53'    DATE_JIS
C     *MDY/         MOVE      CHAR_DATE     DATE_JIS
```

Figure 268. Move Operation with Date

```
C*
C* DATE_USA will contain 12-31-9999
C*
C                   MOVE      *HIVAL        DATE_USA
C*
C*  Execution error, resulting in error code 114.  Year is not in
C*  1940-2039 date range.  DATE_YMD will be unchanged.
C*
C                   MOVE      DATE_USA      DATE_YMD
C*
C* Move a character value representing a *CYMD date to a *USA
C* Date. DATE_USA will contain 08/07/1961 after the move.
C* 0 in *CYMD indicates that the character value does not
C* contain separators
C*    *CYMD0        MOVE      CHAR_NO_SEP   DATE_USA
C*
C* Move a *EUR date field to a numeric field that will
C* represent a *CMDY date.  NUM_DATE2 will contain 2082697
C* after the move.
C    *CMDY          MOVE      DATE_EUR      NUM_DATE2
C*
C* Move a character value representing a *LONGJUL date to
C* a *YMD date.  DATE_YMD will be 39/06/15 after the move.
C    *LONGJUL       MOVE      CHAR_LONGJUL  DATE_YMD
```

Figure 269. Move Operation with Date (continued)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
H* Specify default format for date fields
H DATEFMT(*ISO)
H*
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D date_USA       S               D   DATFMT(*USA)
D datefield      S               D
D timefield      S               T   INZ(T'14.23.10')
D chr_dateA      S              6    INZ('041596')
D chr_dateB      S              7    INZ('0610807')
D chr_time       S              6
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* Move a character value representing a *MDY date to a D(Date) value.
C* *MDY0 indicates that the character date in Factor 2 does not
C* contain separators.
C* datefld will contain 1996-04-15 after the move.
C     *MDY          MOVE      chr_dateA     datefld
C* Move a field containing a T(Time) value to a character value in the
C* *EUR format.  *EUR0 indicates that the result field should not
C* contain separators.
C* chr_time will contain '142310' after the move.
C     *EUR0         MOVE      timefld       chr_time
C*
C* Move a character value representing a *CYMD date to a *USA
C* Date. Date_USA will contain 08/07/1961 after the move.
C* 0 in *CYMD indicates that the character value does not
C* contain separators.
C*
C     *CYMD0        MOVE      chr_dateB     date_USA
```

Figure 270. MOVE Operation with Date and Time, without Separators

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
H*  Control specification DATEDIT format
H*
H DATEDIT(*MDY)
H*
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++
D Jobstart        S               Z
D Datestart       S               D
D Timestart       S               T
D Timebegin       S               T   inz(T'05.02.23')
D Datebegin       S               D   inz(D'1991-09-24')
D TmStamp         S               Z   inz
D*
C* Set the timestamp Jobstart with the job start Date and Time
C *
C * Factor 1 of the MOVE *DATE (*USA = MMDDYYYY) is consistent
C * with the value specified for the DATEDIT keyword on the
C * control specification, since DATEDIT(*MDY) indicates that
C * *DATE is formatted as MMDDYYYY.
C *
C* Note:   It is not necessary to specify factor 1 with *DATE or
C*         UDATE.
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C     *USA          MOVE      *DATE         Datestart
C                   TIME                    StrTime           6 0
C     *HMS          MOVE      StrTime       Timestart
C                   MOVE      Datestart     Jobstart
C                   MOVE      Timestart     Jobstart
C*
C* After the following C specifications are performed, the field
C* stampchar will contain '1991-10-24-05.17.23.000000'.
C*
C* First assign a timestamp the value of a given time+15 minutes and
C* given date + 30 days. Move tmstamp to a character field.
C* stampchar will contain '1991-10-24-05.17.23.000000'.
C*
C                   ADDDUR    15:*minutes   Timebegin
C                   ADDDUR    30:*days      Datebegin
C                   MOVE      Timebegin     TmStamp
C                   MOVE      Datebegin     TmStamp
C                   MOVE      TmStamp       stampchar        26
C* Move the timestamp to a character field without separators. After
C* the move, STAMPCHAR will contain '      19911024051723000000'.
C     *ISO0         MOVE(P)   TMSTAMP       STAMPCHAR0
```

Figure 271. MOVE Operation with Timestamp

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE between graphic and character fields
D*
D char_fld1      S              8A   inz('K1K2K3  ')
D dbcs_fld1      S              4G
D char_fld2      S              8A   inz(*ALL'Z')
D dbcs_fld2      S              3G   inz(G'K1K2K3')
D*
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C* Value of dbcs_fld1 after MOVE operation is 'K1K2K3  '
C* Value of char_fld2 after MOVE operation is 'ZZK1K2K3'
C*
C                   MOVE      char_fld1     dbcs_fld1
C                   MOVE      dbcs_fld2     char_fld2
```

Figure 272. MOVE Between Character and Graphic Fields

MOVE Examples (Part 2): Variable- and Fixed-length Fields

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE from variable to variable length
D* for character fields
D*
D var5a          S              5A   INZ('ABCDE') VARYING
D var5b          S              5A   INZ('ABCDE') VARYING
D var5c          S              5A   INZ('ABCDE') VARYING
D var10a         S             10A   INZ('0123456789') VARYING
D var10b         S             10A   INZ('ZXCVBNM') VARYING
D var15a         S             15A   INZ('FGH') VARYING
D var15b         S             15A   INZ('FGH') VARYING
D var15c         S             15A   INZ('QWERTYUIOPAS') VARYING
C*
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE      var15a        var5a
C* var5a = 'ABFGH' (length=5)
C                   MOVE      var10a        var5b
C* var5b = '56789' (length=5)
C                   MOVE      var5c         var15a
C* var15a = 'CDE' (length=3)
C                   MOVE      var10b        var15b
C* var15b = 'BNM' (length=3)
C                   MOVE      var15c        var10b
C* var10b = 'YUIOPAS' (length=7)
```

Figure 273. MOVE from a Variable-length Field to Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE from variable to fixed length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15          S             15A   INZ('FGH') VARYING
D fix5a          S              5A   INZ('MNOPQ')
D fix5b          S              5A   INZ('MNOPQ')
D fix5c          S              5A   INZ('MNOPQ')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE      var5          fix5a
C* fix5a = 'ABCDE'
C                   MOVE      var10         fix5b
C* fix5b = '56789'
C                   MOVE      var15         fix5c
C* fix5c = 'MNFGH'
```

Figure 274. MOVE from a Variable-length Field to a Fixed-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE from fixed to variable length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15          S             15A   INZ('FGHIJKL') VARYING
D fix5           S              5A   INZ('.....')
D fix10          S             10A   INZ('PQRSTUVWXY')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE      fix10         var5
C* var5 = 'UVWXY' (length=5)
C                   MOVE      fix5          var10
C* var10 = '01234.....' (length=10)
C                   MOVE      fix10         var15
C* var15 = 'STUVWXY' (length=7)
```

Figure 275. MOVE from a Fixed-length Field to a Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE(P) from variable to variable length
D* for character fields
D*
D var5a          S              5A   INZ('ABCDE') VARYING
D var5b          S              5A   INZ('ABCDE') VARYING
D var5c          S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGH') VARYING
D var15b         S             15A   INZ('FGH') VARYING
D var15c         S             15A   INZ('FGH') VARYING
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE(P)   var15a        var5a
C* var5a = '  FGH' (length=5)
C                   MOVE(P)   var10         var5b
C* var5b = '56789' (length=5)
C                   MOVE(P)   var5c         var15b
C* var15b = 'CDE' (length=3)
C                   MOVE(P)   var10         var15c
C* var15c = '789' (length=3)
```

Figure 276. MOVE(P) from a Variable-length Field to a Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE(P) from variable to fixed length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15          S             15A   INZ('FGH') VARYING
D fix5a          S              5A   INZ('MNOPQ')
D fix5b          S              5A   INZ('MNOPQ')
D fix5c          S              5A   INZ('MNOPQ')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE(P)   var5          fix5a
C* fix5a = 'ABCDE'
C                   MOVE(P)   var10         fix5b
C* fix5b = '56789'
C                   MOVE(P)   var15         fix5c
C* fix5c = '  FGH'
```

Figure 277. MOVE(P) from a Variable-length Field to a Fixed-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVE(P) from fixed to variable length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGHIJKLMNOPQR') VARYING
D var15b         S             15A   INZ('FGHIJ') VARYING
D fix5           S              5A   INZ('')
D fix10          S             10A   INZ('PQRSTUVWXY')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVE(P)   fix10         var5
C* var5 = 'UVWXY' (length=5 before and after)
C                   MOVE(P)   fix10         var10
C* var10 = 'PQRSTUVWXY' (length=10 before and after)
C                   MOVE(P)   fix10         var15a
C* var15a = '   PQRSTUVWXY' (length=13 before and after)
C                   MOVE(P)   fix10         var15b
C* var15b = 'UVWXY' (length=5 before and after)
```

Figure 278. MOVE(P) from a Fixed-length Field to a Variable-length Field

MOVE Examples (Part 3)

Table 54. Moving a Character Field to a Date-Time Field. Factor 1 specifies the format of the Factor 2 entry.

		Result Field	
Factor 1	Factor 2 (Character)	Value	DTZ Type
*MDY	11-19-75	75/323	D(*JUL)
*JUL	92/114	23/04/92	D(*DMY)
*YMD	14/01/28	01/28/2014	D*(USA)
*YMD0	140128	01/28/2014	D(*USA)
*USA	12/31/9999	31.12.9999	D(*EUR)
*ISO	2036-05-21	21/05/36	D(*DMY)
*JUL	45/333	11/29/1945	D(*USA)
*MDY/	03/05/33	03.05.33	D(*MDY.)
*CYMD&	121 07 08	08.07.2021	D(*EUR)
*CYMD0	1210708	07,08,21	D(*MDY,)
*CMDY.	107.08.21	21-07-08	D(*YMD-)
*CDMY0	1080721	07/08/2021	D(*USA)
*LONGJUL-	2021-189	08/07/2021	D(*EUR)
*HMS&	23 12 56	23.12.56	T(*ISO)
*USA	1:00 PM	13:00.00	T(*EUR)
*EUR	11.10.07	11:10:07	T(*JIS)
*JIS	14:16:18	14.16.18	T(*HMS.)
*ISO	24:00.00	12:00 AM	*T(*USA)
Blank	1991-09-14-13.12.56.123456	1991-09-14-13.12.56.123456	Z(*ISO)
*ISO	1991-09-14-13.12.56.123456	1991-09-14-13.12.56.123456	Z(*ISO)

Table 55. Moving a Numeric Field to a Date-Time Field. Factor 1 specifies the format of the Factor 2 entry.

		Result Field	
Factor 1	Factor 2 (Numeric)	Value	DTZ Type
*MDY	111975	75/323	D(*JUL)
*JUL	92114	23/04/92	D(*DMY)
*YMD	140128	01/28/2014	D(*USA)
*USA (See note 1.)	12319999	31.12.9999	D(*EUR)
*ISO	20360521	21/05/36	D(*DMY
*JUL	45333	11/29/1945	D(*USA)
*MDY	030533	03.05.33	D(*MDY.)
*CYMD	1210708	08.07.2021	D(*EUR)
*CMDY	1070821	21-07-08	D(*YMD-)
*CDMY	1080721	07/08/2021	D(*USA)

Table 55. Moving a Numeric Field to a Date-Time Field. Factor 1 specifies the format of the Factor 2 entry. (continued)

*LONGJUL	2021189	08/07/2021	D(*EUR)
*USA	*DATE (092195) (See note 3.)	1995-09-21	D(*JIS)
Blank	*DATE (092195) (See note 3.)	1995-09-21	D(*JIS)
*MDY	UDATE (092195) (See note 3.)	21.09.1995	D(*EUR)
*HMS	231256	23.12.56	T(*ISO)
*EUR	111007	11:10:07	T(*JIS)
*JIS	141618	14.16.18	T(*HMS.)
*ISO	240000	12:00 AM	T(*USA)
Blank (See note 4.)	19910914131256123456	1991-09-14-13.12.56.123456	Z(*ISO)

Notes::

1 Time format *USA is not allowed for movement between time and numeric classes.

2 A separator of zero (0) is not allowed in factor 1 for movement between date, time or timestamp fields and numeric classes.

3 For *DATE and UDATE, assume that the job date in the job description is of *MDY format and contains 092195. Factor 1 is optional and will default to the correct format. If factor 2 is *DATE, and factor 1 is coded, it must be a 4-digit year date format. If factor 2 is UDATE, and factor 1 is coded, it must be a 2-digit year date format.

4 For moves of timestamp fields, factor 1 is optional. If it is coded it must be *ISO or *ISO0.

MOVE Examples (Part 4)

Table 56. Moving a Date-Time Field to a Character Field

	Factor 2		
Factor 1 Entry	Value	DTZ Type	Result Field (Character)
*JUL	11-19-75	D(*MDY-)	75/323
*DMY-	92/114	D(*JUL)	23-04-92
*USA	14/01/28	D(*YMD)	01/28/2014
*EUR	12/31/9999	D(*USA)	31.12.9999
*DMY,	2036-05-21	D(*ISO)	20,05,36
*USA	45/333	D(*JUL)	11/29/1945
*USA0	45/333	D(*JUL)	11291945
*MDY&	03/05/33	D(*MDY)	03 05 33
*CYMD,	03 07 08	D(*DMY)	1080721
*CMDY	21-07-08	D(*YMD-)	107/08/21
*CDMY-	07/08/2021	D(*USA)	108-07-21
*LONGJUL&	08/07/2021	D(*EUR)	2021 189
*ISO	23 12 56	T(*HMS&)	23.12.56
*EUR	11:00 AM	T(*USA)	11.00.00
*JIS	11.10.07	T(*EUR)	11:10:07
*HMS,	14:16:18	T(*JIS)	14,16,18
*USA	24.00.00	T(*ISO)	12:00 AM
Blank	2045-10-27-23.34.59.123456	Z(*ISO)	2045-10-27-23.34.59.123456

Table 57. Moving a Date-Time Field to a Numeric Field

	Factor 2		
Factor 1 Entry	Value	DTZ Type	Result Field (Numeric)
*JUL	11-19-75	D(*MDY-)	75323
*DMY-	92/114	D(*JUL)	230492
*USA	14/01/28	D(*YMD)	01282014
*EUR	12/31/9999	D(*USA)	31129999
*DMY	2036-05-21	D(*ISO)	210536
*USA	45/333	D(*JUL)	11291945
*MDY&	03/05/33	D(*MDY)	030533
*CYMD,	03 07 08	D(*MDY&)	1080307
*CMDY	21-07-08	D(*YMD-)	1070821
*CDMY-	07/08/2021	D(*USA)	1080721
*LONGJUL&	08/07/2021	D(*EUR)	2021189
*ISO	231256	T(*HMS&)	231256
*EUR	11:00 AM	T(*USA)	110000
*JIS	11.10.07	T(*EUR)	111007
*HMS,	14:16:18	T(*JIS)	141618

Table 57. Moving a Date-Time Field to a Numeric Field (continued)

*ISO	2045-10-27-23.34.59.123456	Z(*ISO)	20451027233459123456

The following table shows examples of moving a date-time fields to date-time fields. Assume that the initial value of the timestamp is: *1985-12-03-14.23.34.123456.*

Table 58. Moving Date-Time Fields to Date-Time Fields

	Factor 2		Result Field	
Factor 1	Value	DTZ Type	Value	DTZ Type
N/A	1986-06-24	D(*ISO)	86/06/24	D(*YMD)
N/A	23 07 12	D(*DMY&)	23.07.2012	D(*EUR)
N/A	11:53 PM	T(USA)	23.53.00	T(*EUR)
N/A	19.59.59	T(*HMS)	19:59:59	T(*JIS)
N/A	1985-12-03-14.23.34.123456	Z(*ISO.)	1985-12-03-14.23.34.123456	Z(*ISO)
N/A	75.06.30	D(*YMD)	1975-06-30-14.23.34.123456	Z(*ISO)
N/A	09/23/2234	D(*USA)	2234-09-23-14.23.34.123456	Z(*ISO)
N/A	18,45,59	T(*HMS,)	1985-12-03-18.45.59.000000	Z(*ISO)
N/A	2:00 PM	T(*USA)	1985-12-03-14.00.00.000000	Z(*ISO)
N/A	1985-12-03-14.23.34.123456	Z(*ISO.)	12/03/85	D(*MDY)
N/A	1985-12-03-14.23.34.123456	Z(*ISO.)	12/03/1985	D(*USA)
N/A	1985-12-03-14.23.34.123456	Z(*ISO.)	14:23:34	T(*HMS)
N/A	1985-12-03-14.23.34.123456	Z(*ISO)	02:23 PM	T(*USA)

MOVE Examples (Part 5)

The following table shows examples of moving a date field to a character field. The result field is larger than factor 2. Assume that factor 1 contains *ISO and that the result field is defined as:

```
D     Result_Fld        20S   INZ('ABCDEFGHIJabcdefghij')
```

Table 59. Moving a Date Field to a Character Field

	Factor 2		Value of Result Field after move operation
Operation Code	Value	DTZ Type	
MOVE	11 19 75	D(*MDY&)	'ABCDEFGHIJ1975-11-19'
MOVE(P)	11 19 75	D(*MDY&)	' 1975-11-19'
MOVEL	11 19 75	D(*MDY&)	'1975-11-19abcdefghij'
MOVEL(P)	11 19 75	D(MDY&)	'1975-11-19 '

The following table shows examples of moving a time field to a numeric field. The result field is larger than factor 2. Assume that Factor 1 contains *ISO and that the result field is defined as:

```
D     Result_Fld        20S   INZ(11111111111111111111)
```

Table 60. Moving a Time Field to a Numeric Field

	Factor 2		
Operation Code	Value	DTZ Type	Value of Result Field after move operation
MOVE	9:42 PM	T(*USA)	11111111111111214200
MOVE(P)	9:42 PM	T(*USA)	00000000000000214200
MOVEL	9:42 PM	T(*USA)	21420011111111111111
MOVEL(P)	9:42 PM	T(*USA)	21420000000000000000

Table 61. Moving a Numeric field to a Time Field. Factor 2 is larger than the result field. The highlighted portion shows the part of the factor 2 field that is moved.

		Result Field	
Operation Code	Factor 2	DTZ Type	Value
MOVE	11:**12:13:14**	T(*EUR)	12.13.14
MOVEL	**11:12:13**:14	T(*EUR)	11.12.13

Table 62. Moving a Numeric field to a Timestamp Field. Factor 2 is larger than the result field. The highlighted portion shows the part of the factor 2 field that is moved.

		Result Field	
Operation Code	Factor 2	DTZ Type	Value
MOVE	123406**18230323123420123456**	Z(*ISO)	1823-03-23-12.34.21.123456
MOVEL	**12340618230323123420**123456	Z(*ISO)	1234-06-18-23-.03.23.123420

MOVEA (Move Array)

Free-Form Syntax	(not allowed — use %SUBARR or one or more String Operations)

Code	Factor 1	Factor 2	Result Field	Indicators		
MOVEA (P)		Source	Target	+	–	ZB

The MOVEA operation transfers character, graphic, UCS-2, or numeric values from factor 2 to result field. (Certain restrictions apply when moving numeric values.)

You can use the MOVEA operation to:
- Move several contiguous array elements to a single field
- Move a single field to several contiguous array elements
- Move contiguous array elements to contiguous elements of another array.

Movement of data starts with the first element of an array if the array is not indexed or with the element specified if the array is indexed. The movement of data ends when the last array element is moved or filled. When the result field contains the indicator array, the cross-reference listing contains all the indicators affected by the MOVEA operation.

Factor 2 or the result field must contain an array. The array can be packed, binary, zoned, graphic or a character array. Factor 2 and the result field cannot specify the same array even if the array is indexed.

Note: For character, graphic, UCS-2, and numeric MOVEA operations, you can specify the P operation extender to pad the result from the right.

For more information, see “Array Operations” on page 351, “Move Operations” on page 368, or “Date Operations” on page 359.

Character, Graphic, and UCS-2 MOVEA Operations

Both factor 2 and the result field must be defined as character, graphic, or UCS-2. Graphic or UCS-2 CCSIDs must be the same, unless, in the case of graphic fields, CCSID(*GRAPH: *IGNORE) was specified on the control specification. Movement of data ends when the number of characters moved equals the shorter length of the fields specified by factor 2 and the result field.

The MOVEA operation could end in the middle of an array element.

Variable-length arrays are not allowed.

Numeric MOVEA Operations

The data that is moved between fields and array elements must have the same length. Factor 2 and the result field must contain numeric fields, numeric array elements, or numeric arrays. At least one must be an array or array element. The numeric types can be binary, packed decimal, or zoned decimal. The numeric types do not need to be the same between factor 2 and the result field.

Factor 2 can contain a numeric literal if the result field contains a numeric array or numeric array-element:
- The numeric literal cannot contain a decimal point.

- The length of the numeric literal cannot be greater than the element length of the array element specified in the result field.

Decimal positions are ignored during the move. Numeric values are not converted to account for the differences in the defined number of decimal positions.

If the result field contains a numeric array, factor 2 cannot contain the figurative constants *BLANK, *ALL, *ON, and *OFF.

Zoned Decimal MOVEA Operations

To move a zoned decimal format array:
- Define the numeric array as a subfield of a data structure
- Redefine the numeric array in the data structure as as a character array.

You can then use MOVEA the same way as you would for character array moves.

Specifying Figurative Constants with MOVEA

To move a figurative constant, the length of the constant must be equal to the portion of the array that is specified. For figurative constants in numeric arrays, the element boundaries are ignored except for the sign that is put in each element array. For example:
- MOVEA *BLANK ARR(X)

  Beginning with element X, the remainder of ARR contains blanks.
- MOVEA *ALL'XYZ'ARRR(X)

  ARR has 4-byte character elements. Element boundaries are ignored. Beginning with element X, the remainder of the array contains 'XYZXYZXYZYXZ...'.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY
C*  Array-to-array move.  No indexing; different length array,
C*  same element length.
```

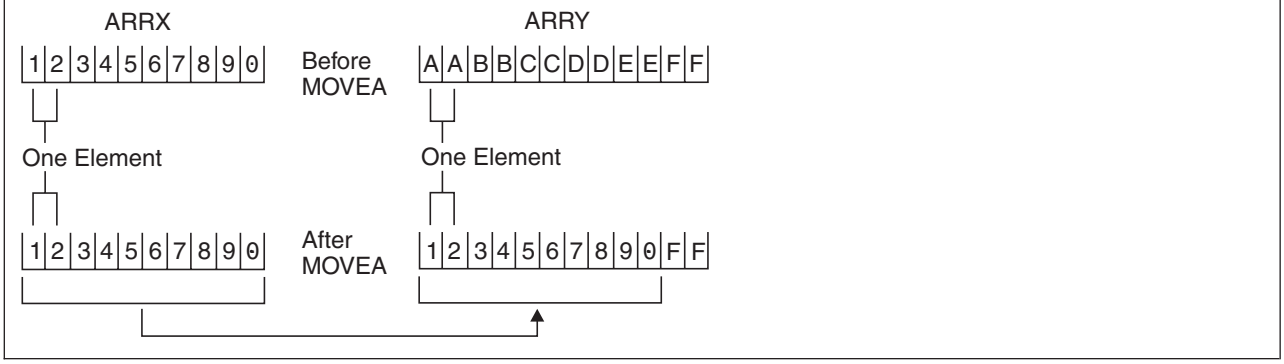


Figure 279. Array to Array - Different Array Length, Same Element Length

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY(3)
C*  Array-to-array move with index result field.
```

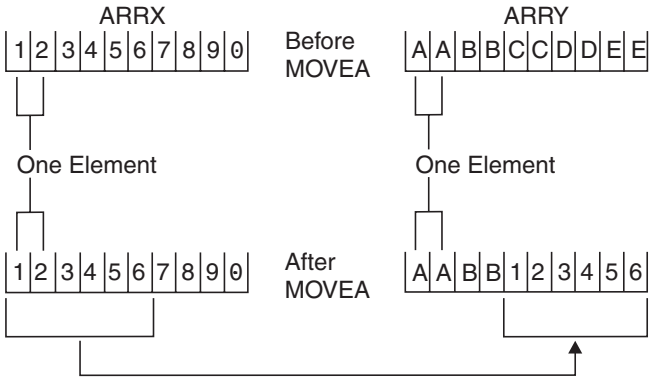


Figure 280. Array to Array - Indexed Result Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY
C*  Array-to-array move, no indexing and different length array
C*  elements.
```

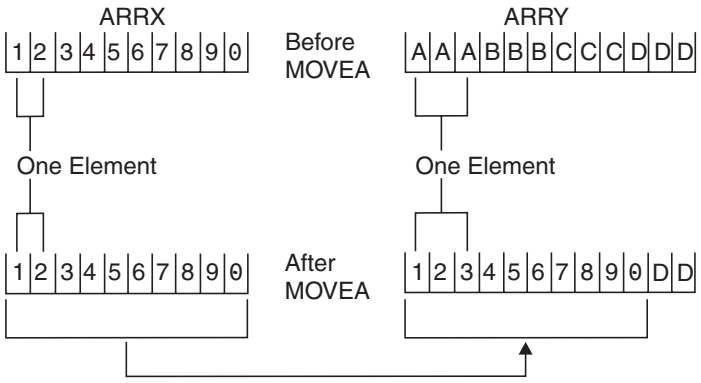


Figure 281. Array to Array - Different Length Array Elements

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX(4)       ARRY
C*  Array-to-array move, index factor 2 with different length array
C*  elements.
```

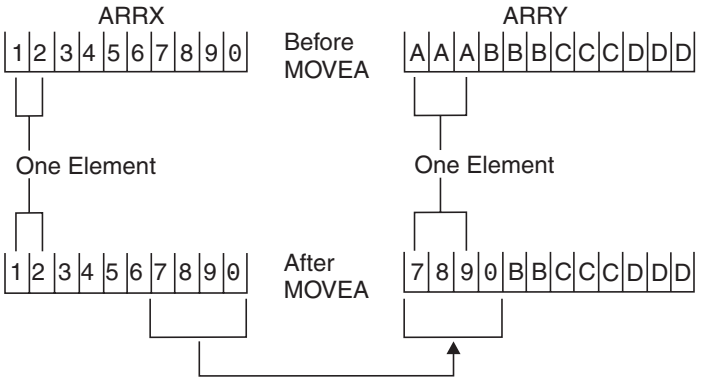


Figure 282. Array to Array - Indexed Factor 2

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     FIELDA        ARRY
C*  Field-to-array move, no indexing on array.
```

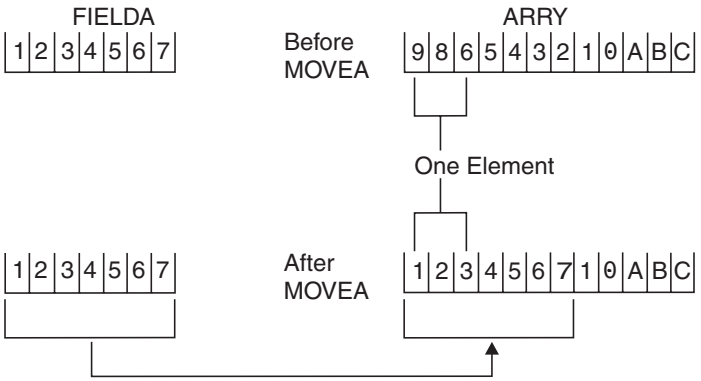


Figure 283. Array to Array - No Indexing

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* In the following example, N=3.  Array-to-field move with variable
C* indexing.
C                   MOVEA     ARRX(N)       FIELD
C*
```

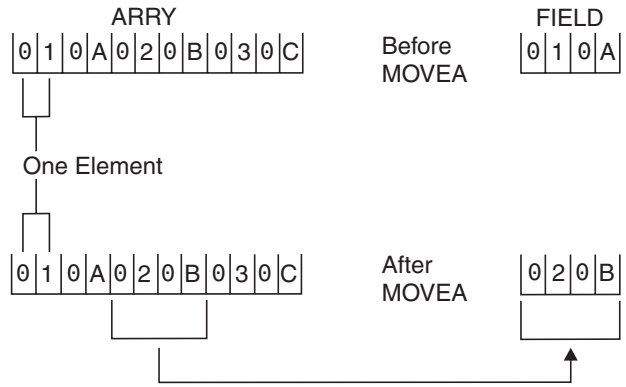


Figure 284. Array to field - Variable indexing

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRB          ARRZ
C*
C* An array-to-array move showing numeric elements.
```

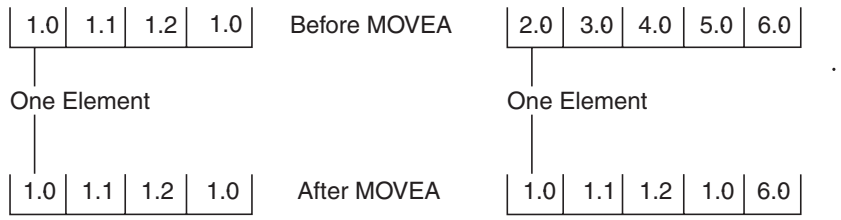


Figure 285. Array to array - Numeric Elements

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRX          ARRY
C*  Array-to-array move with padding.  No indexing; different length
C*  array with same element length.
```

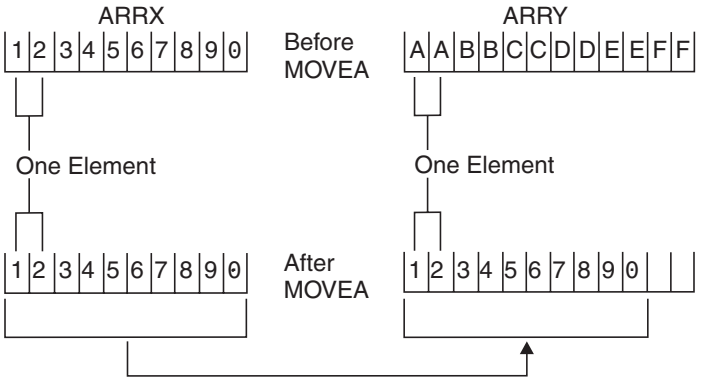


Figure 286. Array to array - With Padding

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRB          ARRZ
C*
C* An array-to-array move showing numeric elements with padding.
```

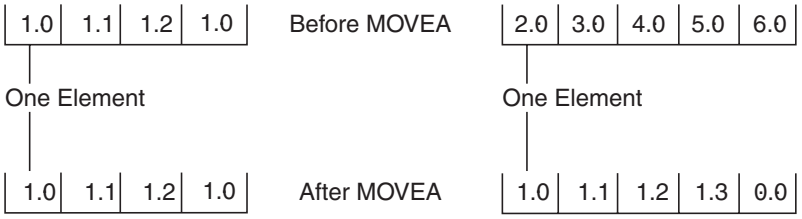


Figure 287. Array to array - Numeric Elements with Padding

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRX(3)       ARRY
C*  Array-to-array move with padding.  No indexing; different length
C*  array with different element length.
```

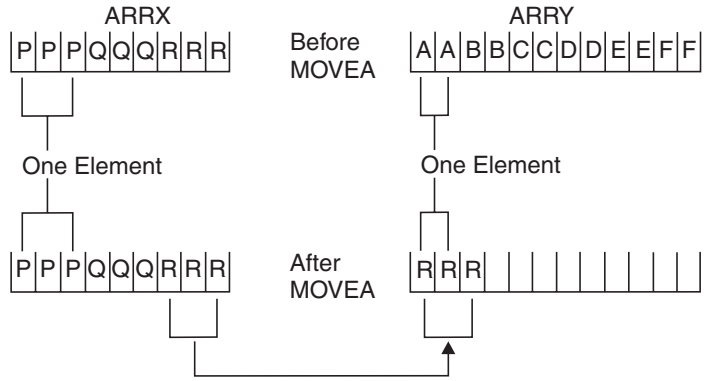


Figure 288. Array to array - With Padding, No Indexing

MOVEL (Move Left)

Free-Form Syntax	not allowed - use EVAL, or built-in functions such as %CHAR, %DATE, %DEC , %DECH, %GRAPH, %INT, %INTH, %TIME, %TIMESTAMP , %UCS2, %UNS, or %UNSH

Code	Factor 1	Factor 2	Result Field	Indicators		
MOVEL (P)	Data Attributes	Source field	Target field	+	–	ZB

The MOVEL operation transfers characters from factor 2 to the result field. Moving begins with the leftmost character in factor 2.

If factor 1 is specified, it must contain a date or time format. This specifies the format of the character or numeric field that is the source or target of the operation.

You cannot specify resulting indicators if the result field is an array. You can specify them if the result field is an array element, or a nonarray field.

If the source or target is a character field, you may optionally indicate the separator following the format in factor 1. Only separators that are valid for that format are allowed.

If factor 2 is *DATE or UDATE and the result is a Date field, factor 1 is not required. If factor 1 contains a date format it must be compatible with the format *DATE or UDATE in factor 2 as specified by the DATEDIT keyword on the control specification.

If factor 2 is longer than the result field, the excess rightmost characters of factor 2 are not moved. If the result field is longer than factor 2, the excees rightmost characters in the result field are unchanged, unless padding is specified.

Float numeric fields and literals are not allowed as Factor 2.

If factor 2 is UCS-2 and the result field is character, or if factor 2 is character and the result field is UCS-2, the number of characters moved is variable. For example, five UCS-2 characters can convert to:
- Five single-byte characters
- Five double-byte characters
- A combination of single-byte and double-byte characters

Note: When data is moved to a numeric field, the sign (+ or –) of the result field is retained except when factor 2 is as long as or longer than the result field. In this case, the sign of factor 2 is used as the sign of the result field.

The following sections summarize the rules for the MOVEL operation based on the length of factor 2 and the result field.

Factor 2 is the Same Length as the Result Field

Factor 2 and the result field are the same length:

- If factor 2 and the result field are both numeric, the sign is moved into the rightmost position.
- If factor 2 and the result field are both character, all characters are moved.
- If factor 2 is numeric and the result field is character, the sign is moved into the rightmost position.
- If factor 2 is character and the result field is numeric, a minus zone is moved into the rightmost position of the result field if the zone from the rightmost position of factor 2 is a minus zone. However, if the zone from the rightmost position of factor 2 is not a minus zone, a positive zone is moved into the rightmost position of the result field. Digit portions are converted to their corresponding numeric characters. If the digit portions are not valid digits, a data exception error occurs.
- If factor 2 and the result field are both graphic or UCS-2, all graphic or UCS-2 characters are moved.
- If factor 2 is graphic and the result field is character, all graphic characters are moved.
- If factor 2 is character and the result field is graphic, all characters are moved.

Factor 2 is Longer than the Result Field

Factor 2 is longer than the result field:

- If factor 2 and the result field are both numeric, the sign from the rightmost position of factor 2 is moved into the rightmost position of the result field.
- If factor 2 is numeric and the result field is character, the result field contains only numeric characters. Only the number of characters needed to fill the result field are moved.
- If factor 2 is character and the result field is numeric, a minus zone is moved into the rightmost position of the result field if the zone from the rightmost position of factor 2 is a minus zone. However, if the zone from the rightmost position of factor 2 is not a minus zone, a positive zone is moved into the rightmost position of the result field. Other result field positions contain only numeric characters.
- If factor 2 and the result field are both graphic or UCS-2, only the number of graphic or UCS-2 characters needed to fill the result field are moved.
- If factor 2 is graphic and the result field is character, the graphic data is truncated.
- If factor 2 is character and the result is graphic, the character data is truncated.

Note: The excess rightmost characters of factor 2 are not moved. If the result field is longer than factor 2, the excess rightmost characters in the result field are unchanged, unless padding is specified.

Factor 2 is Shorter than the Result Field

Factor 2 is shorter than the result field:

- If factor 2 is either numeric or character and the result field is numeric, the digit portion of factor 2 replaces the contents of the leftmost positions of the result field. The sign in the rightmost position of the result field is not changed.
- If factor 2 is either numeric or character and the result field is character data, the characters in factor 2 replace the equivalent number of leftmost positions in the result field. No change is made in the zone of the rightmost position of the result field.

Factor 2 is Shorter than the Result Field and P is Specified

If factor 2 is shorter than the result field, and P is specified in the operation extender field:

- The move is performed as described in "Factor 2 is Shorter than the Result Field" on page 627
- The result field is padded from the right.

When moving **variable-length** character, graphic, or UCS-2 data, the variable-length field works in exactly the same way as a fixed-length field with the same current length. For examples, see Figures 293 to 298.

For further information on the MOVEL operation, see "Move Operations" on page 368, "Date Operations" on page 359, or "Conversion Operations" on page 358.

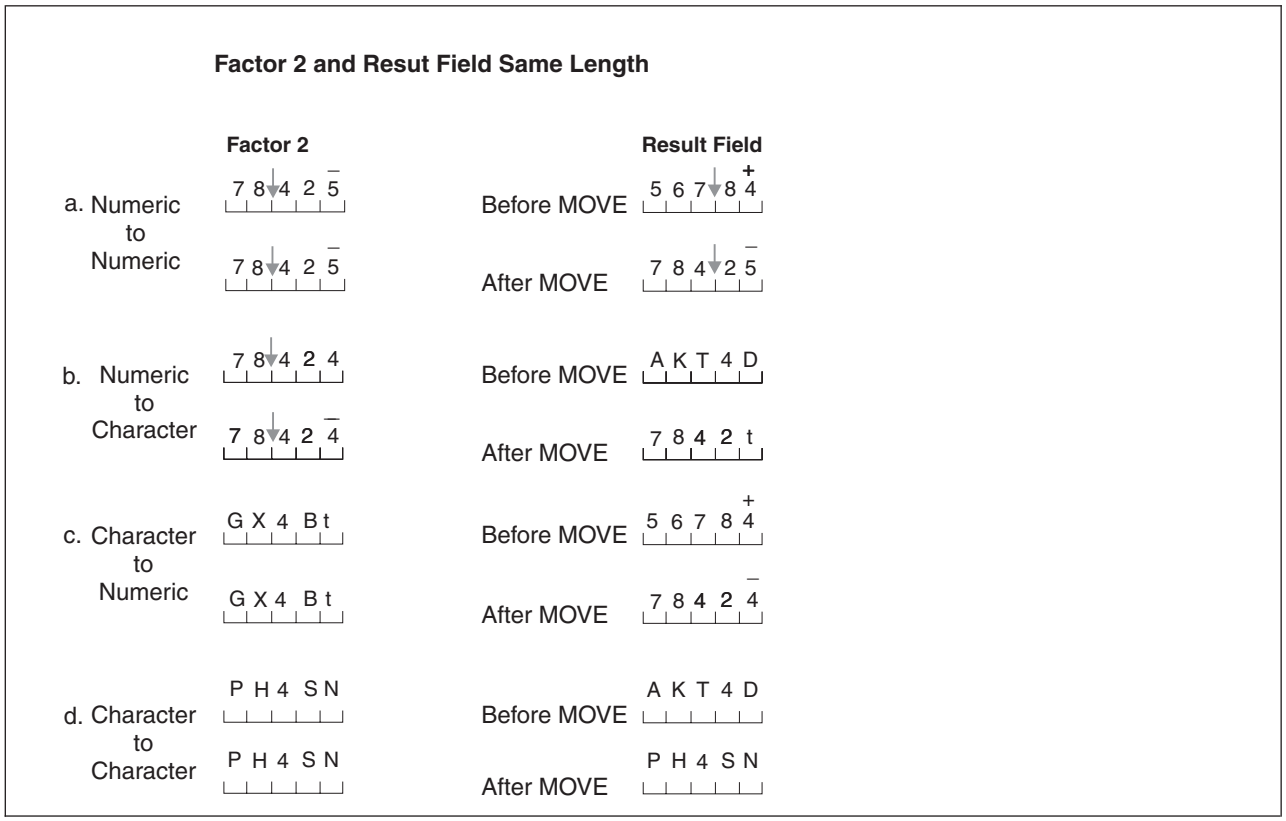


Figure 289. Factor 2 and the Result Field are the Same Length

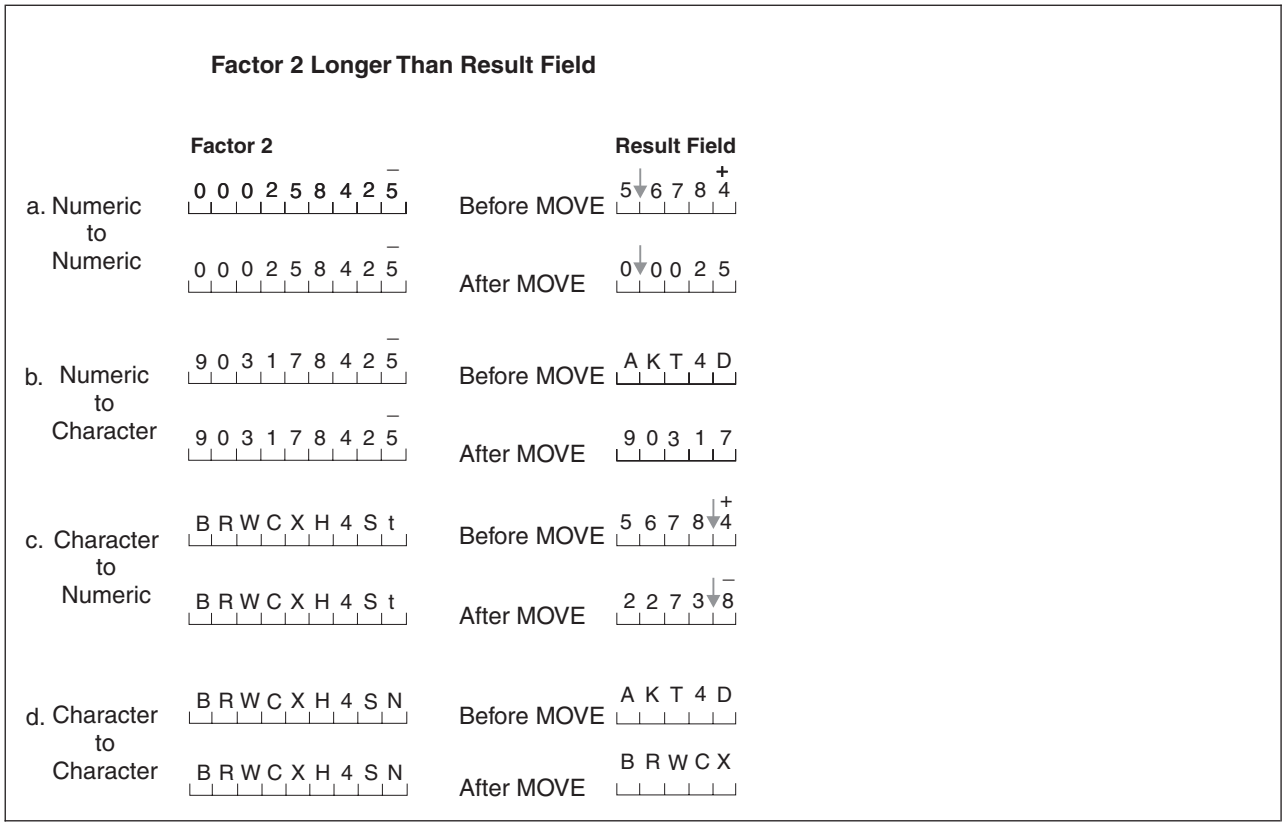


Figure 290. Factor 2 is Longer than the Result Field

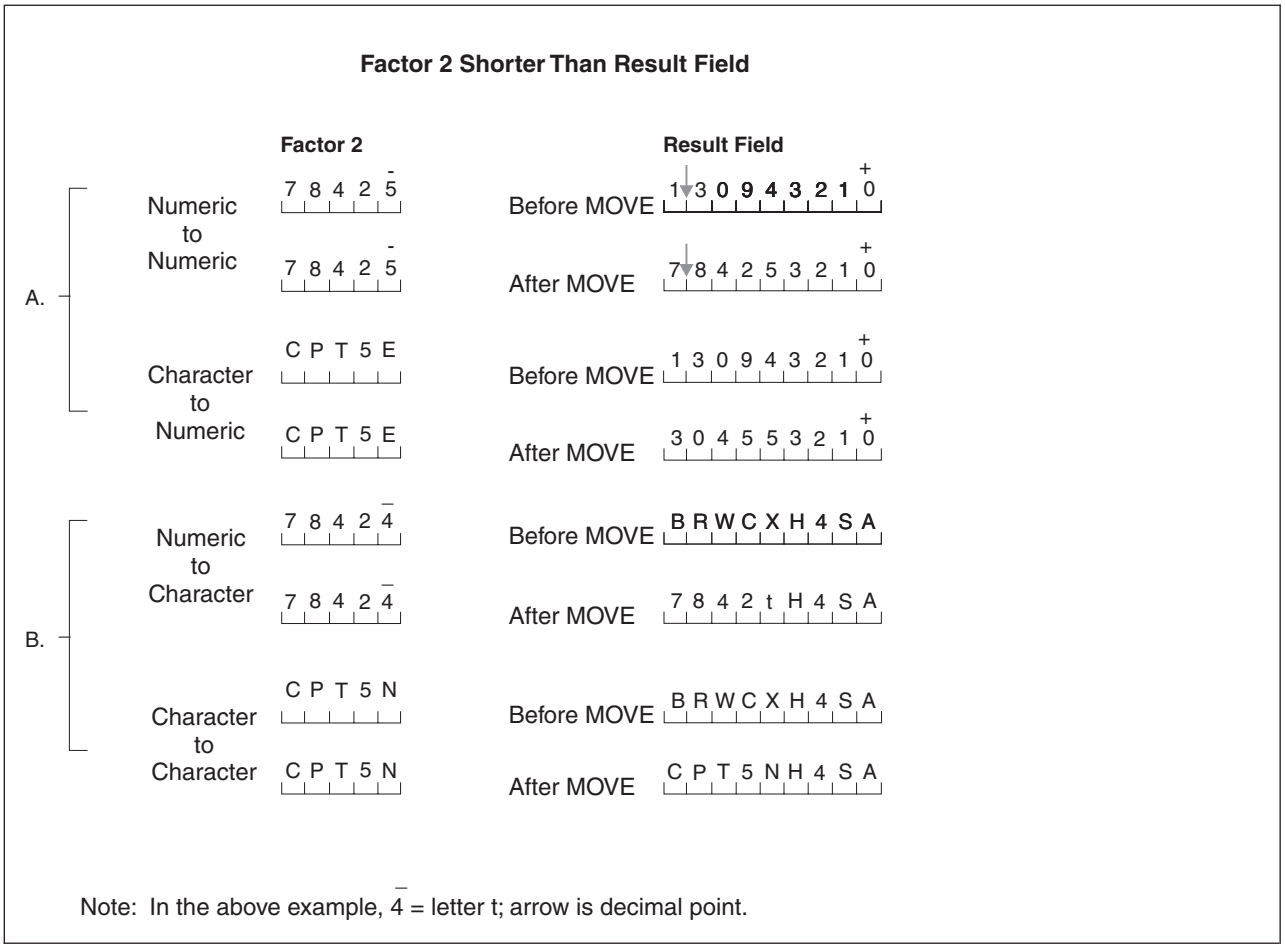


Figure 291. Factor 2 is Shorter than the Result Field

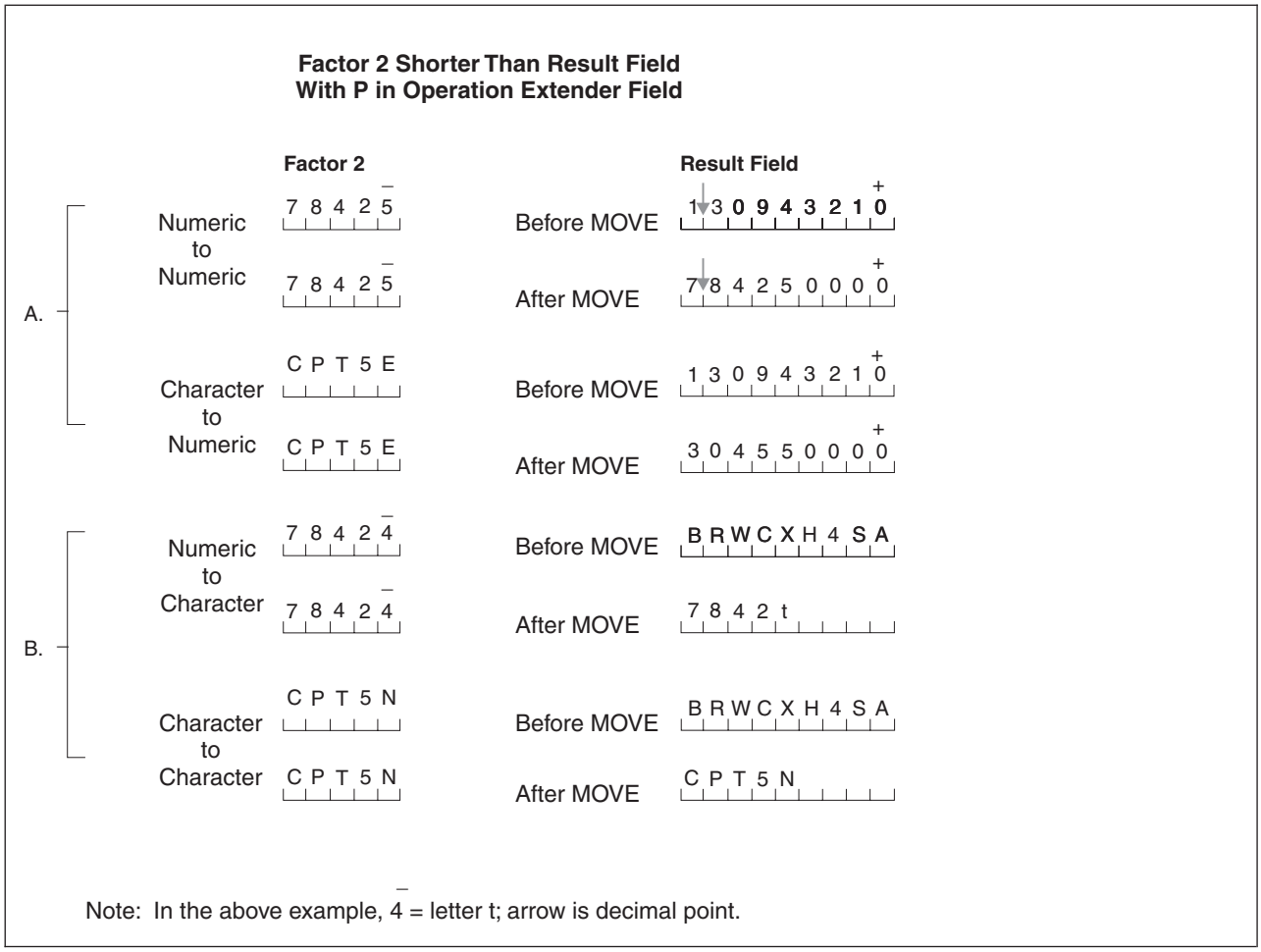


Figure 292. Factor 2 is Shorter than the Result with P Specified

MOVEL Examples: Variable-length / Fixed-length Moves

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL from variable to variable length
D* for character fields
D*
D var5a          S              5A   INZ('ABCDE') VARYING
D var5b          S              5A   INZ('ABCDE') VARYING
D var5c          S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGH') VARYING
D var15b         S             15A   INZ('FGH') VARYING
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL     var15a        var5a
C* var5a = 'FGHDE' (length=5)
C                   MOVEL     var10         var5b
C* var5b = '01234' (length=5)
C                   MOVEL     var5c         var15a
C* var15a = 'ABC' (length=3)
C                   MOVEL     var10         var15b
C* var15b = '012' (length=3)
```

Figure 293. MOVEL: Variable-length Field to Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL from variable to fixed length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15          S             15A   INZ('FGH') VARYING
D fix5a          S              5A   INZ('MNOPQ')
D fix5b          S              5A   INZ('MNOPQ')
D fix5c          S              5A   INZ('MNOPQ')
D fix10          S             10A   INZ('')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL     var5          fix5a
C* fix5a = 'ABCDE'
C                   MOVEL     var10         fix5b
C* fix5b = '01234'
C                   MOVEL     var15         fix5c
C* fix5c = 'FGHPQ'
```

Figure 294. MOVEL: Variable-length Field to Fixed-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL from fixed to variable length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGHIJKLMNOPQR') VARYING
D var15b         S             15A   INZ('WXYZ') VARYING
D fix10          S             10A   INZ('PQRSTUVWXY')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL     fix10         var5
C* var5 = 'PQRST' (length=5)
C                   MOVEL     fix10         var10
C* var10 = 'PQRSTUVWXY' (length=10)
C                   MOVEL     fix10         var15a
C* var15a = 'PQRSTUVWXYPQR' (length=13)
C                   MOVEL     fix10         var15b
C* var15b = 'PQRS' (length=4)
```

Figure 295. MOVEL: Fixed-length Field to Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL(P) from variable to variable length
D* for character fields
D*
D var5a          S              5A   INZ('ABCDE') VARYING
D var5b          S              5A   INZ('ABCDE') VARYING
D var5c          S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGH') VARYING
D var15b         S             15A   INZ('FGH') VARYING
D var15c         S             15A   INZ('FGHIJKLMN') VARYING
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL(P)  var15a        var5a
C* var5a = 'FGH  ' (length=5)
C                   MOVEL(P)  var10         var5b
C* var5b = '01234' (length=5)
C                   MOVEL(P)  var5c         var15b
C* var15b = 'ABC' (length=3)
C                   MOVEL(P)  var15a        var15c
C* var15c = 'FGH      ' (length=9)
```

Figure 296. MOVEL(P): Variable-length Field to Variable-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL(P) from variable to fixed length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15          S             15A   INZ('FGH') VARYING
D fix5a          S              5A   INZ('MNOPQ')
D fix5b          S              5A   INZ('MNOPQ')
D fix5c          S              5A   INZ('MNOPQ')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL(P)  var5          fix5a
C* fix5a = 'ABCDE'
C                   MOVEL(P)  var10         fix5b
C* fix5b = '01234'
C                   MOVEL(P)  var15         fix5c
C* fix5c = 'FGH  '
```

Figure 297. MOVEL(P): Variable-length Field to a Fixed-length Field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++
D*
D* Example of MOVEL(P) from fixed to variable length
D* for character fields
D*
D var5           S              5A   INZ('ABCDE') VARYING
D var10          S             10A   INZ('0123456789') VARYING
D var15a         S             15A   INZ('FGHIJKLMNOPQR') VARYING
D var15b         S             15A   INZ('FGH') VARYING
D fix5           S             10A   INZ('.....')
D fix10          S             10A   INZ('PQRSTUVWXY')
D*
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
C*
C                   MOVEL(P)  fix10         var5
C* var5 = 'PQRST' (length=5)
C                   MOVEL(P)  fix5          var10
C* var10 = '.....     ' (length=10)
C                   MOVEL(P)  fix10         var15a
 * var15a = 'PQRSTUVWXY   ' (length=13)
C                   MOVEL(P)  fix10         var15b
C* var15b = 'PQR' (length=3)
```

Figure 298. MOVEL(P): Fixed-length field to Variable-length Field

MULT (Multiply)

Free-Form Syntax	(not allowed - use the * or *= operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
MULT (H)	Multiplicand	Multiplier	Product	+	–	Z

If factor 1 is specified, factor 1 is multiplied by factor 2 and the product is placed in the result field. If factor 1 is not specified, factor 2 is multiplied by the result field and the product is placed in the result field.

Factor 1 and factor 2 must be numeric, and each can contain an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be large enough to hold the product. Use the following rule to determine the maximum result field length: result field length equals the length of factor 1 plus the length of factor 2. The result field must be numeric, but cannot be a named constant or literal. You can specify half adjust to have the result rounded.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the MULT operation.

MVR (Move Remainder)

Free-Form Syntax	(not allowed - use the %REM built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
MVR			Remainder	+	–	Z

The MVR operation moves the remainder from the previous DIV operation to a separate field named in the result field. The MVR operation must immediately follow the DIV operation. If you use conditioning indicators, the MVR operation must be specified immediately after the DIV operation. The result field must be numeric and can contain an array, array element, subfield, or table name.

Leave sufficient room in the result field if the DIV operation uses factors with decimal positions. The number of significant decimal positions is the greater of:
- The number of decimal positions in factor 1 of the previous divide operation
- The sum of the decimal positions in factor 2 and the result field of the previous divide operation.

The maximum number of whole number positions in the remainder is equal to the whole number of positions in factor 2 of the previous divide operation.

The sign (+ or -) of the remainder is the same as the dividend (factor 1).

You cannot specify half adjust on a DIV operation that is immediately followed by an MVR operation. The MVR operation cannot be used if the previous divide operation has an array specified in the result field. Also, the MVR operation cannot be used if the previous DIV operation has at least one float operand.

“Arithmetic Operations” on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the MVR operation.

OCCUR (Set/Get Occurrence of a Data Structure)

Free-Form Syntax	(not allowed - use the %OCCUR built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
OCCUR (E)	Occurrence value	Data structure	Occurrence value	_	ER	_

The OCCUR operation specifies the occurrence of the data structure that is to be used next within a program.

If a data structure with multiple occurrences or a subfield of that data structure is specified in an operation, the first occurrence of the data structure is used until an OCCUR operation is specified. After an OCCUR operation is specified, the occurrence of the data structure that was established by the OCCUR operation is used.

If factor 1 is specified, it must contain a numeric, zero decimal position literal, field name, named constant, or a data structure name. Factor 1 sets the occurrence of the data structure specified in factor 2. If factor 1 is not specified, the value of the current occurrence of the data structure in factor 2 is placed in the result field during the OCCUR operation.

If factor 1 is a data structure name, it must be a multiple occurrence data structure. The current occurrence of the data structure in factor 1 is used to set the occurrence of the data structure in factor 2.

Factor 2 must be the name of a multiple occurrence data structure.

If the result field is specified, it must be a numeric field name with no decimal positions. The value of the current occurrence of the data structure specified in factor 2, after being set by any value or data structure that is optionally specified in factor 1, is placed in the result field.

Note: At least one of factor 1 or the result field must be specified.

If the occurrence is outside the valid range set for the data structure, an error occurs, and the occurrence of the data structure in factor 2 remains the same as before the OCCUR operation was processed.

To handle OCCUR exceptions (program status code 122), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

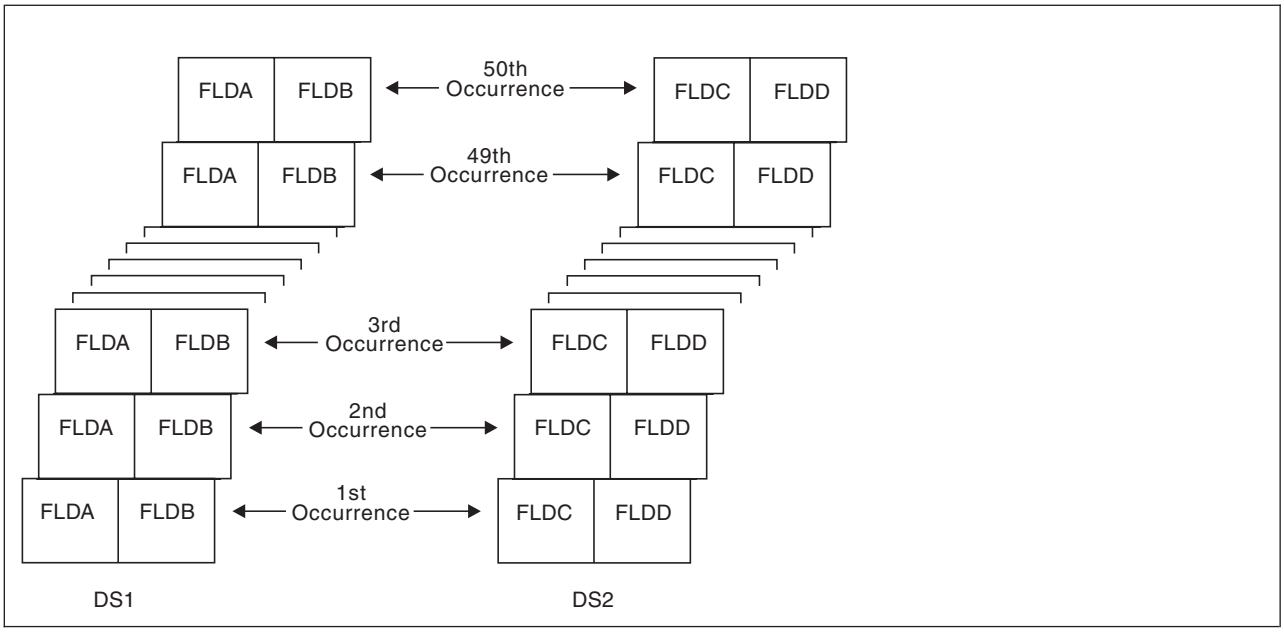


Figure 299. OCCUR Operation Example

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D*
D*  DS1 and DS2 are multiple occurrence data structures.
D*  Each data structure has 50 occurrences.
D DS1             DS                  OCCURS(50)
D  FLDA                    1      5
D  FLDB                    6     80
D*
D DS2             DS                  OCCURS(50)
D  FLDC                    1      6
D  FLDD                    7     11
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*  DS1 is set to the third occurrence.  The subfields FLDA
C*  and FLDB of the third occurrence can now be used.  The MOVE
C*  and Z-ADD operations change the contents of FLDA and FLDB,
C*  respectively, in the third occurrence of DS1.
C
C     3             OCCUR     DS1
C                   MOVE      'ABCDE'       FLDA
C                   Z-ADD     22            FLDB
C*
C*  DS1 is set to the fourth occurrence.  Using the values in
C*  FLDA and FLDB of the fourth occurrence of DS1, the MOVE
C*  operation places the contents of FLDA in the result field,
C*  FLDX, and the Z-ADD operation places the contents of FLDB
C*  in the result field, FLDY.
C
C     4             OCCUR     DS1
C                   MOVE      FLDA          FLDX
C                   Z-ADD     FLDB          FLDY
C*
C*  DS1 is set to the occurrence specified in field X.
C*  For example, if X = 10, DS1 is set to the tenth occurrence.
C     X             OCCUR     DS1
C*
C*  DS1 is set to the current occurrence of DS2.  For example, if
C*  the current occurrence of DS2 is the twelfth occurrence, DSI
C*  is set to the twelfth occurrence.
C     DS2           OCCUR     DS1
```

Figure 300. OCCUR Operation Example

```
C*
C*  The value of the current occurrence of DS1 is placed in the
C*  result field, Z.  Field Z must be numeric with zero decimal
C*  positions.  For example, if the current occurrence of DS1
C*  is 15, field Z contains the value 15.
C                   OCCUR     DS1           Z
C
C*  DS1 is set to the current occurrence of DS2.  The value of the
C*  current occurrence of DS1 is then moved to the result field,
C*  Z.  For example, if the current occurrence of DS2 is the fifth
C*  occurrence, DS1 is set to the fifth occurrence.  The result
C*  field, Z, contains the value 5.
C
C     DS2           OCCUR     DS1           Z
C*
C*  DS1 is set to the current occurrence of X.  For example, if
C*  X = 15, DS1 is set to the fifteenth occurrence.
C*  If X is less than 1 or greater than 50,
C*  an error occurs and %ERROR is set to return '1'.
C*  If %ERROR returns '1', the LR indicator is set on.
C
C     X             OCCUR (E) DS1
C                   IF        %ERROR
C                   SETON                                         LR
C                   ENDIF
```

Figure 301. OCCUR Operation Example

ON-ERROR (On Error)

Free-Form Syntax	ON-ERROR {*exception-id1* {:*exception-id2*...}}

Code	Factor 1	Extended Factor 2
ON-ERROR		List of exception IDs

You specify which error conditions the on-error block handles in the list of exception IDs (*exception-id1*:*exception-id2*...). You can specify any combination of the following, separated by colons:

nnnnn	A status code
***PROGRAM**	Handles all program-error status codes, from 00100 to 00999
***FILE**	Handles all file-error status codes, from 01000 to 09999
***ALL**	Handles both program-error and file-error codes, from 00100 to 09999. This is the default.

Status codes outside the range of 00100 to 09999, for example codes from 0 to 99, are not monitored for. You cannot specify these values for an on-error group. You also cannot specify any status codes that are not valid for the particular version of the compiler being used.

If the same status code is covered by more than one on-error group, only the first one is used. For this reason, you should specify special values such as *ALL after the specific status codes.

Any errors that occur within an on-error group are not handled by the monitor group. To handle errors, you can specify a monitor group within an on-error group.

When all the statements in an on-error block have been processed, control passes to the statement following the ENDMON statement.

For an example of the ON-ERROR statement, see “MONITOR (Begin a Monitor Group)” on page 602.

For more information, see “Error-Handling Operations” on page 362.

OPEN (Open File for Processing)

Free-Form Syntax	OPEN{(E)} *file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
OPEN (E)		file-name		_	ER	_

The explicit OPEN operation opens the file named in the *file-name* operand. The file can either be a local file or an OS/400 file. If the file is defined as a local file and if it does not exist when the OPEN operation occurs, the local file is created. Remote files must exist when the OPEN operation occurs, otherwise it is not created.

The file cannot be a table file. To allow your program to control when the file should be opened, specify the USROPN keyword on the file description specifications. See Chapter 17, "File Description Specifications," on page 237 for more information on the USROPN keyword.

If a file is opened and then closed by the CLOSE operation, the file can be reopened with the OPEN operation. The USROPN keyword on the file description specification is not required. If the USROPN keyword is not specified on the file description specification, the file is opened at program initialization. If an OPEN operation is specified for a file that is already open, an error occurs.

Multiple OPEN operations in a program to the same file are valid as long as the file has been closed prior to the OPEN operation.

If a resulting indicator is specified in positions 73 and 74 on a fixed-format syntax calculation, it is set on when an error occurs during the OPEN operation.

To handle OPEN exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

For more information, see "File Operations" on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++
FEXCEPTN   O    E             DISK    REMOTE USROPN
FFILEX     IF   E             DISK    REMOTE
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The explicit OPEN operation opens the EXCEPTN file for
C*  processing if indicator 97 is on and indicator 98 is off.
C*  Note that the EXCEPTN file on the file description
C*  specifications has the USROPN keyword specified.
C*  %ERROR is set to return '1' if the OPEN operation fails.
C*
C                   IF        *in97 and not *in98
C                   OPEN(E)   EXCEPTN
C                   IF        not %ERROR
C                   WRITE     ERREC
C                   ENDIF
C                   ENDIF
C*
C*  FILEX is opened at program initialization.  The explicit
C*  CLOSE operation closes FILEX before control is passed to RTNX. Upon
C*  return, the OPEN operation reopens the file.  Because the USROPN
C*  keyword is not specified for FILEX, the file is opened at
C*  program initialization
C*
C                   CLOSE     FILEX
C                   CALL      'RTNX'
C                   OPEN      FILEX
```

Figure 302. OPEN Operation with CLOSE Operation

ORxx (Or)

Free-Form Syntax	(not allowed - use the OR operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
ORxx	Comparand	Comparand				

The ORxx operation is optional with the DOUxx, DOWxx, IFxx, WHENxx, and ANDxx operations. ORxx is specified immediately following a DOUxx, DOWxx, IFxx, WHENxx, ANDxx or ORxx statement. Use ORxx to specify a more complex condition for the DOUxx, DOWxx, IFxx, and WHENxx operations. Conditioning indicator entries (positions 9 through 11) are not allowed.

Factor 1 and factor 2 must contain a literal, a named constant, a figurative constant, a table name, an array element, a data structure name, or a field name. Factor 1 and factor 2 must be of the same type. The comparison of factor 1 and factor 2 follows the same rules as those given for the compare operations.

"Compare Operations" on page 357 describes the general rules for specifying compare operations.

Figure 242 on page 558 shows examples of ORxx and ANDxx operations with a DOUxx operation.

For more information, see "Structured Programming Operations" on page 376.

OTHER (Otherwise Select)

Free-Form Syntax	OTHER

Code	Factor 1	Factor 2	Result Field	Indicators		
OTHER						

The OTHER operation begins the sequence of operations to be processed if no WHENxx or WHEN condition is satisfied in a SELECT group. The sequence ends with the ENDSL or END operation.

Rules to remember when using the OTHER operation:
- The OTHER operation is optional in a SELECT group.
- Only one OTHER operation can be specified in a SELECT group.
- No WHENxx or WHEN operation can be specified after an OTHER operation in the same SELECT group.
- The sequence of calculation operations in the OTHER group can be empty; the effect is the same as not specifying an OTHER statement.
- Conditioning indicator entries (positions 9 through 11) are not allowed.

For more information on select groups, see "SELECT (Begin a Select Group)" on page 676 and "WHENxx (When True Then Select)" on page 714.

For more information, see "Structured Programming Operations" on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* Example of a SELECT group with WHENxx and OTHER.  If X equals 1,
C* do the operations in sequence 1; if X does not equal 1 and Y
C* equals 2, do the operations in sequence 2.  If neither
C* condition is true, do the operations in sequence 3.
C*
C                   SELECT
C     X             WHENEQ    1
C*
C* Sequence 1
C*
C                      :
C                      :
C     Y             WHENEQ    2
C*
C* Sequence 2
C*
C                      :
C                      :
C                   OTHER
C*
C* Sequence 3
C*
C                      :
C                      :
C                   ENDSL
```

Figure 303. OTHER Operation

OUT (Write a Data Area)

Free-Form Syntax	OUT{(E)} {*LOCK} *data-area-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
OUT (E)	*LOCK	data-area-name		_	ER	_

The OUT operation updates the data area specified in the *data-area-name* operand. To specify a data area as the *data-area-name* operand of an OUT operation, you must ensure two things:

- The data area must also be specified in the result field of a *DTAARA DEFINE statement, or defined using the DTAARA keyword on the Definition specification.
- The data area must have been locked previously by a *LOCK IN statement or it must have been specified as a data area data structure by a U in position 23 of the definition specifications. (RPG implicitly retrieves and locks data area data structures at program initialization.)

You can specify the optional reserved word *LOCK. When *LOCK is specified, the data area remains locked after it is updated. When *LOCK is not specified, the data area is unlocked after it is updated.

If a data area is locked, it can be read but not updated by other programs.

The *data-area-name* operand must be either the name of the data area or the reserved word *DTAARA. When *DTAARA is specified, all data areas defined in the program are updated. If an error occurs when one or more data areas are updated (for example, if you specify an OUT operation to a data area that has not been locked by the program), an error occurs on the OUT operation and the RPG exception/error handling routine receives control. If a message is issued, it will identify the data area in error.

If a resulting indicator is specified in positions 73 and 74, it is set on when an error occurs during the OUT operation.

To handle OUT exceptions (program status codes 401-421, 431, or 432), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Positions 71-72 and 75-76 must be blank.

For a description of general rules, see "Data-Area Operations" on page 358. See Figure 259 on page 590 for an example of the OUT operation.

PARM (Identify Parameters)

Free-Form Syntax	(not allowed - use "Prototypes and Parameters" on page 71 and CALLP)

Code	Factor 1	Factor 2	Result Field	Indicators		
PARM	Target field	Source field	Parameter			

The PARM operation defines the parameters that compose a parameter list (PLIST). PARM operations must immediately follow the PLIST, CALL, CALLB, or START operation they refer to. PARM statements must be in the order expected by the called program or function. The maximum number of parameters that can be specified is:

- For a CALL operation, up to 255 parameters can be specified
- For a CALLB, START (start component), PLIST operation, up to 399 parameters can be specified.

Figure 304 on page 651 illustrates the PARM operation.

Note: If you are using CALLP to call a local program, parameters are defined by specifying the prototype on the definition specification. "Positions 24-25 (Type of Definition)" on page 261 and "OPTIONS(*OMIT *VARSIZE *STRING *TRIM *RIGHTADJ)" on page 283 describe how to specify parameters for CALLP operations.

If factor 1 is specified, it must be the same type as the result field. If the target field is variable-length, its length will be set to the length of the value of the source field. It cannot be a literal or a named constant. It can be blank if the result field contains the name of a multiple-occurrence data structure.

If factor 2 is specified, it must be the same type as the result field. It can be blank if the result field contains the name of a multiple-occurrence data structure.

If parameter type-checking is important for the application, you should define a prototype and procedure interface definition of the call interface, rather than use the PLIST and PARM operations.

The result field must contain the name of a field, a data structure, or an array:

- If an array is specified, the area defined for the array is passed to the called program or procedure
- If a data structure with multiple occurrences is passed to the called program, all occurrences of the data structure are passed as a single field. However, if a subfield of a multiple occurrence data structure is specified in the result field, only the current occurrence of the subfield is passed to the called program or procedure.

The result field cannot contain a UCS-2 parameter unless a host program is being called.

For non-*ENTRY PLIST PARM operations, the result field can contain the name of an array element or *OMIT (for the CALLB only). If *OMIT is specified, factor 1 and factor 2 must be blank.

For *ENTRY PLIST PARM operations, the result field cannot contain the following:

- *IN, *INxx, *IN(xx), *OMIT
- A label, literal, or a named constant
- A data-area name or a data-area data structure name
- A globally initialized data structure, a data structure with initialized subfields, or a data structure with a compile time array as a subfield
- A table name
- Fields or data structures defined with the keyword BASED
- An array element
- A data-structure subfield name
- The name of a compile-time array
- The name of a program status or file information data structure (INFDS)
- UCS-2 parameters are not allowed.

Note: A field name can be specified only once in an *ENTRY PLIST.

Conditioning indicator entries (positions 9 through 11) are not allowed.

For more information, see "Call Operations" on page 353 or "Declarative Operations" on page 362.

General Rules about Parameters

The storage location for each parameter field is in the calling program or procedure. The address of the storage location of the result field on a PARM operation is passed to the called program. If the called program or procedure changes the value of a parameter, it changes the data at that storage location. When control returns to the calling program or procedure, the parameter in the calling program or procedure (that is, the result field) has changed. Even if the called program or procedure ends in error after it changes the value of a parameter, the changed value exists in the calling program or procedure. To preserve the information passed to the called program or procedure for later use, specify in factor 2 the name of the field that contains the information you want to pass to the called program or procedure. Factor 2 is copied into the result field, and the storage address of the result field is passed to the called program or procedure.

Because the parameter fields are accessed by address, not field name, the calling and called parameters do not have to use the same field names for fields that are passed. The attributes of the corresponding parameter fields in the calling and called programs or procedures should be the same. If they are not, undesirable results may occur.

Passing Parameters with CALL, CALLB, and START

When a CALL, CALLB, or START (starting a component) operation runs, the following occurs:

1. In the calling program, the contents of factor 2 of a PARM operation are copied into the result field of the same PARM operation. If the result field of CALLB is *OMIT, a null address is passed to the called procedure.
2. After the called program receives control and after any normal program initialization, the contents of the result field of a PARM operation are copied into the factor 1 field of the same PARM operation.
3. When control is returned to the calling program, the contents of factor 2 of a PARM operation are copied into the result field of the same PARM operation. This move does not occur if the called program ends abnormally.
4. For the START operation, control is returned to the calling program as soon as the target component is initialized (after *INZSR processing has completed). For

the remainder of the target component's life, the parameter is accessible and can be modified by both the source and target components.

5. Upon return to the calling program, the contents of the result field of a PARM operation are copied into the factor 1 field of the same PARM operation. This move does not occur if the called program ends abnormally or if an error occurs on the call operation.

PLIST (Identify a Parameter List)

Free-Form Syntax	(not allowed - use "Prototypes and Parameters" on page 71 and CALLP)

Code	Factor 1	Factor 2	Result Field	Indicators		
PLIST	PLIST name					

The PLIST operation defines a unique symbolic name for a parameter list to be specified in a CALL, CALLB, CALLP, or START operation. The PLIST operation must be immediately followed by at least one PARM operation.

Factor 1 must contain the name of the parameter list. If the parameter list is the entry parameter list, factor 1 must contain *ENTRY. Only one *ENTRY parameter list can be specified in a program or called function. A parameter list is ended when an operation other than PARM is encountered.

If parameter type checking is important for the application, you should deine a prototype and procedure interface definition for the call interface, rather than use the PLIST and PARM operations.

Conditioning indicator entries (positions 9 through 11) are not allowed.

For more information, see "Call Operations" on page 353 or "Declarative Operations" on page 362.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  In the calling program, the CALL operation calls PROG1 and
C*  allows PROG1 to access the data in the parameter list fields.
C                   CALL     'PROG1'        PLIST1
C*
C*  In the second PARM statement, when CALL is processed, the
C*  contents of factor 2, *IN27, are placed in the result field,
C*  BYTE.  When PROG1 returns control, the contents of the result
C*  field, BYTE, are placed in the factor 1 field, *IN30.  Note
C*  that factor 1 and factor 2 entries on a PARM are optional.
C*
C     PLIST1        PLIST
C                   PARM                    Amount            5 2
C     *IN30         PARM      *IN27         Byte              1
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   CALLB     'PROG2'
.
.
.
C*  In this example, the PARM operations immediately follow a
C*  CALLB operation instead of a PLIST operation.
C                   PARM                    Amount            5 2
C     *IN30         PARM      *IN27         Byte              1
.
.
.
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*  In the called function, PROG2,  *ENTRY in factor 1 of the
C*  PLIST statement identifies it as the entry parameter list.
C*  When control transfers to PROG2, the contents of the result
C*  fields (FieldC and FieldG) of the parameter list are placed in
C*  the factor 1 fields (FieldA and FieldD).  When the called function
C*  returns, the contents of the factor 2 fields of the parameter
C*  list (FieldB and FieldE) are placed in the result fields (FieldC
C*  and FieldG).  All of the fields are defined elsewhere in called
C*  function.
C     *ENTRY        PLIST
C     FieldA        PARM      FieldB        FieldC
C     FieldD        PARM      FieldE        FieldG
```

Figure 304. PLIST/PARM Operations

POST (Post)

Free-Form Syntax	POST{(E)} *file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
POST (E)		file-name	INFDS name	_	ER	_

The POST operation puts information in a file information data structure (INFDS).

For remote files, this structure contains the following information:
- File Feedback Information
- Open Feedback Information
- Input/Output Feedback Information and Device Dependent Feedback Information

For local files, this structure contains the File Feedback Information.

Specify the name of a file in the *file-name* operand. Information for this file is posted in the INFDS associated with this file.

In free-form syntax, you must specify a *file-name* and cannot specify an INFDS name. In traditional syntax, you can specify a *file-name*, an INFDS name, or both.
- If you do not specify an INFDS name, the INFDS associated with this file using the INFDS keyword in the file specification will be used.
- If you do not specify an INFDS name in traditional syntax, you must specify the data structure name that has been used in the INFDS keyword for the file specification in the result field; information from the associated file in the file specification will be posted.

If the *file-name* operand is specified, it can either be a local file or an OS/400 file. This file must be opened prior to a POST operation. Information for this file is posted in its associated INFDS.

If a file is opened for multiple member processing, the Open Feedback Information is updated when an input operation such as READ, READP, READE, or READPE causes a new member to be opened.

If the input records are blocked and there is no POST operation in the application, the current key and relative record number are copied in the Input/Output Feedback Information. If the input records are blocked and there is a POST operation in the application, then the Input/Output Feedback Information is updated with the key and relative record number of the current record in the block.

If a resulting indicator is specified in positions 73 and 74, it is set on when an error occurs during the POST operation.

To handle POST exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

For more information, see "File Operations" on page 363.

READ (Read a Record)

Free-Form Syntax	READ{(EN)} *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READ (E N)		name (file or record format)	data-structure	_	ER	EOF
READ (E)		Window name		_	ER	_

The READ operation reads data from a file, a record format, or from a window. The file can be a remote OS/400 file or a local file.

Reading from a File

The *name* operand is required and must be the name of a full procedural file or record format.

A record format name is allowed only with an externally described file (E in position 22 of the file description specifications). It may be the case that a READ-by-format-name operation will receive a different format from the one you specified in the *name* operand. If so, your READ operation ends in error.

If the *data-structure* operand is specified, the record is read directly into the data structure. If *name* refers to a program-described file (identified by an F in position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

If a READ operation is successful, the file is positioned at the next record that satisfies the read. If there is an error or an end of file condition, you must reposition the file (using a CHAIN, SETLL, or SETGT operation).

If the file is an update disk file, the operation extender N can be specified to indicate that no lock should be placed on the record when it is read.

Note: Locking is not supported for local files.

To handle READ exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 to signal whether an end of file occurred on the READ operation. The indicator is either set on (an EOF condition) or off every time the READ operation is performed. This information can also be obtained from the %EOF built-in function, which returns '1' if an EOF condition occurs and '0' otherwise. The file must be repositioned after an EOF condition, in order to process any further successful sequential operations (for example, READ or READP) to the file.

See "Database Null Value Support" on page 137 for information on reading records with null-capable fields.

READ (Read a Record)

For more information, see “File Operations” on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  READ retrieves the next record from the file FILEA, which must
C*  be a full procedural file.
C*
C*  %EOF is set to return '1' if an end of file occurs on READ,
C*  or if an end of file has occurred previously and the file
C*  has not been repositioned.  When %EOF returns '1',
C*  the program will leave the loop.
C*
C                   DOW       '1'
C                   READ      FILEA
C                   IF        %EOF
C                   LEAVE
C                   ENDIF
C*
C*  READ retrieves the next record of the type REC1 (factor 2)
C*  from an externally described file.  (REC1 is a record format
C*  name.)  Indicator 64 is set on if end of file occurs on READ,
C*  or if it has occurred previously and the file has not been
C*  repositioned.  When indicator 64 is set on, the program
C*  will leave the loop.  The N operation code extender
C*  indicates that the record is not locked.
C*
C                   READ(N)   REC1                                  64
C   64              LEAVE
C                   ENDDO
```

Figure 305. READ Operation for Files

Reading from a Window

Windows are handled as externally described files. The window name is treated as a record format name.

If a window name is specified by the *name* operand, the READ operation gets the attributes of the combination box, check box, entry field, radio button, and static text parts on the window. The attribute for entry parts is TEXT. The attribute for static text parts is LABEL.

When a window is read, get attribute operations are performed on all the static text and entry field parts. The values are stored in corresponding fields. After the READ operation, the values stored in the fields match the values on the display. If there are many static text and entry fields, use the READ operation rather than multiple GETATRs. For example, if window INVENTORY contains the entry field parts ENT0000B and ENT0000C a READ of the window performs the equivalent to the following:

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments++++++
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++Comments++++++
C                   EVAL      ENT0000B = %GETATR('INVENTORY':'ENT0000B':'TEXT')
C                   EVAL      ENT0000C = %GETATR('INVENTORY':'ENT0000C':'TEXT')
```

Figure 306. READ Operation for Windows

READC (Read Next Changed Record)

Free-Form Syntax	READC{(E)} *subfile-name* {*subfile-index*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READC (E)		subfile-name	subfile-index	_	ER	EOF

The READC operation obtains the next changed record in the subfile part.

The *subfile-name* operand must be the name of a subfile part.

If the *subfile-index* operand is specified, it must be a numeric field name with no decimal positions. The relative record number of the retrieved record is placed in the *subfile-index* operand.

To handle READC exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that will be set on when there are no more changed records in the subfile. This information can also be obtained from the %EOF built-in function, which returns '1' if there are no more changed records in the subfile and '0' otherwise.

If an end of file indicator (EOF) is specified, it is set on when there are no more changed records in the subfile. If the operation was not successful, the fields in the program remain unchanged.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++
F* SUBCUST is a subfile part which displays a list of records from
F* the CUSINFO file.
F*
FCUSINFO   UF   E             DISK    REMOTE
F
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C* The subfile SUBCUST has been loaded with the records from the
C* CUSINFO file. If there are any changes in any one of the records
C* displayed in the subfile, the READC operation will read the changed
C* records one by one in the do while loop.
C* The corresponding record in the CUSINFO file will be located
C* with the CHAIN operation and will be updated with the changed
C* field.
C* SCUSNO, SCUSNAM, SCUSADR, and SCUSTEL are fields defined in the
C* subfile. CUSNAM, CUSADR, and CUSTEL are fields defined in a
C* record, CUSREC, which is defined in the file CUSINFO.
C*
C                   READC     SUBCUST
C                   DOW       %EOF = *OFF
C     SCUSNO        CHAIN (E) CUSINFO
C* Update the record only if the record is found in the file.
C                   :
C                   IF        NOT %ERROR
C                   EVAL      CUSNAM = SCUSNAM
C                   EVAL      CUSADR = SCUSADR
C                   EVAL      CUSTEL = SCUSTEL
C                   UPDATE    CUSREC
C                   ENDIF
C                   READC (E) SUBCUST
C                   ENDDO
```

Figure 307. READC Example

READE (Read Equal Key)

Free-Form Syntax	READE{(ENHMR)} *search-arg*|*KEY *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READE (E N)	search-arg	name (file or record format)	data-structure	_	ER	EOF

The READE operation retrieves the next sequential record from a full procedural file (identified by an F in position 18 of the file description specifications) if the key of the record matches the search argument. If the key of the record does not match the search argument, an EOF condition occurs, and the record is *not* returned to the program. An EOF condition also applies when end of file occurs.

The READE operation can only be used with OS/400 files.

The search argument, *search-arg*, identifies the record to be retrieved. The *search-arg* operand is optional in traditional syntax but is required in free-form syntax. *search-arg* can be:

- A field name, a literal, a named constant, or a figurative constant.
- A KLIST name for an externally described file.
- A list of key values enclosed in parentheses. See Figure 227 on page 531 for an example of searching using a list of key values.
- %KDS to indicate that the search arguments are the subfields of a data structure. See the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451 for an illustration of search arguments in a data structure.
- *KEY or (in traditional syntax only) no value. If the full key of the next record is equal to that of the current record, the next record in the file is retrieved. The full key is defined by the record format or file specified in *name*.

Graphic and UCS-2 keys must have the same CCSID.

If the file being read is defined as update, a temporary lock on the next record is requested and the search argument is compared to the key of that record. If the record is already locked, the program must wait until the record is available before obtaining the temporary lock and making the comparison. If the comparison is unequal, an EOF condition occurs, and the temporary record lock is removed. If no lock ('N' operation extender) is specified, a temporary lock is not requested.

The *name* operand must be the name of the file or record format to be retrieved. A record format name is allowed only with an externally described file (identified by an E in position 22 of the file description specifications.)

If the *data-structure* operand is specified, the record is read directly into the data structure. If *name* refers to a program-described file (identified by an F in position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

To handle READE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that will be set on if an EOF condition occurs: that is, if a record is not found with a key equal to the search argument or if an end of file is encountered. This information can also be obtained from the %EOF built-in function, which returns '1' if an EOF condition occurs and '0' otherwise.

If the READE operation is not successful, the fields in the program remain unchanged and the file must be repositioned (for example, using CHAIN, SETLL or SETGT). *START and *END can be used to position the file. For more information on file positioning, see "File Positioning" on page 6.

A READE with *search-arg* specified that immediately follows an OPEN operation or an EOF condition, retrieves the first record in the file if the key of the record matches the search argument. A READE with no *search-arg* specified that immediately follows an OPEN operation or an EOF condition, results in an error. The error indicator in positions 73 and 74, if specified, is set on or the 'E' extender, checked with %ERROR, if specified, is set on. No further I/O operations can be issued against the file until it is successfully closed and reopened.

See "Database Null Value Support" on page 137 for information on reading records with null capable fields.

For more information, see "File Operations" on page 363.

Note: Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  With Factor 1 Specified...
C*
C*  The READE operation retrieves the next record from the file
C*  FILEA and compares its key to the search argument, KEYFLD.
C*  The %EOF built-in function is set to return '1' if KEYFLD is
C*  not equal to the key of the record read or if end of file
C*  is encountered.
C*
C     KEYFLD        READE     FILEA
C*
C*  The READE operation retrieves the next record of the type REC1
C*  from an externally described file and compares the key of the
C*  record read to the search argument, KEYFLD.  (REC1 is a record
C*  format name.)  Indicator 56 is set on if KEYFLD is not equal to
C*  the key of the record read or if end of file is encountered.
C*
C     KEYFLD        READE     REC1                                   56
C*
C*  With No Factor 1 Specified...
C*
C*  The READE operation retrieves the next record in the access
C*  path from the file FILEA if the key value is equal to
C*  the key value of the record at the current cursor position.
C*  If the key values are not equal, %EOF is set to return '1'.
C                   READE     FILEA
C*
C*  The READE operation retrieves the next record in the access
C*  path from the file FILEA if the key value equals the key value
C*  of the record at the current position.  REC1 is a record format
C*  name.  Indicator 56 is set on if the key values are unequal.
C*  N indicates that the record is not locked.
C                   READE(N)  REC1                                   56
```

Figure 308. READE Operation

READP (Read Prior Record)

Free-Form Syntax	READP{(EN)} *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READP (E N)		name (file or record format)	data-structure	_	ER	BOF

The READP operation reads the prior record from a full procedural file (identified by an F in position 18 of the file description specifications).

The *name* operand must be the name of a file or record format to be read. A record format name is allowed only with an externally described file. If a record format name is specified, the record retrieved is the first prior record of the specified type. Intervening records are bypassed.

If the *data-structure* operand is specified, the record is read directly into the data structure. If *name* refers to a program-described file (identified by an F in position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

If the READP operation is successful, the file is positioned at the previous record that satisfies the read.

If the file being read is an update disk file, the operation extender N can be specified to indicate that no lock should be placed on the record when it is read.

To handle READP exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that will be set on when no prior records exist in the file (beginning of file condition). This information can also be obtained from the %EOF built-in function, which returns '1' if a BOF condition occurs and '0' otherwise.

You must reposition the file (for example, using a CHAIN, SETLL or SETGT operation) after an error or BOF condition to process any further successful sequential operations (for example, READ or READP). *START and *END can be used to position the file. For more information on file positioning, see "File Positioning" on page 6

See "Database Null Value Support" on page 137 for information on reading records with null-capable fields.

For more information, see "File Operations" on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The READP operation reads the prior record from FILEA.
C*  The %EOF built-in function is set to return '1' if beginning
C*  of file is encountered.  When %EOF returns '1', the program
C*  branches to the label BOF specified in the GOTO operation.
C*
C                   READP     FILEA
C                   IF        %EOF
C                   GOTO      BOF
C                   ENDIF
C*
C*  The READP operation reads the next prior record of the type
C*  REC1 from an externally described file.  (REC1 is a record
C*  format name.)  Indicator 72 is set on if beginning of file is
C*  encountered during processing of the READP operation.  When
C*  indicator 72 is set on, the program branches to the label BOF
C*  specified in the GOTO operation.
C                   READP     PREC1                                  72
C   72              GOTO      BOF
C*
C     BOF           TAG
```

Figure 309. READP Operation

READPE (Read Prior Equal)

Free-Form Syntax	READPE{(ENHMR)} *search-arg*|*KEY *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READPE (E N)	search-arg	name (file or record format)	data-structure	_	ER	BOF

The READPE operation retrieves the next prior sequential record from a full procedural file (identified by an F in position 18 of the file description specifications) if the key of the record matches the search argument. If the key of the record does not match the search argument, a BOF condition occurs, and the record is *not* returned to the program. A BOF condition also applies when beginning of file occurs.

The READPE operation can only be used with OS/400 files.

The search argument, *search-arg*, identifies the record to be retrieved. The *search-arg* operand is optional in traditional syntax but required in free-form syntax. *search-arg* can be:

- A field name, a literal, a named constant, or a figurative constant.
- A KLIST name for an externally described file.
- A list of key values enclosed in parentheses. See Figure 227 on page 531 for an example of searching using a list of key values.
- %KDS to indicate that the search arguments are the subfields of a data structure. See the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451 for an illustration of search arguments in a data structure.
- *KEY or (in traditional syntax only) no value. If the full key of the next prior record is equal to that of the current record, the next prior record in the file is retrieved. The full key is defined by the record format or file used in factor 2.

Graphic and UCS-2 keys must have the same CCSID.

The *name* operand must be the name of the file or record format to be retrieved. A record format name is allowed only with an externally described file (identified by an E in position 22 of the file description specifications).

If the *data-structure* operand is specified, the record is read directly into the data structure. If *name* refers to a program-described file (identified by an F in position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

If the file being read is an update disk file, you can specify the operation extender N to indicate that no lock should be placed on the record when it is read.

To handle READPE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that will be set on if a BOF (begining of file) condition occurs: that is, if a record is not found with a key equal to the search argument or if a beginning of file is encountered. This information can also be obtained from the %EOF built-in function, which returns '1' if a BOF condition occurs and '0' otherwise.

If a READPE operation is not successful, you must reposition the file: for example, using a CHAIN, SETGT, or SETLL operation.

Note: If the file being read is defined as update, a temporary lock on the prior record is requested and the search argument is compared to the key of that record. If the record is already locked, the program must wait until the record is available before obtaining the temporary lock and making the comparison. If the comparison is unequal, a BOF condition occurs, and the temporary record lock is removed. If no lock ('N' operation extender) is specified, a temporary lock is not requested.

A READPE with the *search-arg* operand specified that immediately follows an OPEN operation or a BOF condition returns BOF. A READPE with **no** *search-arg* specified that immediately follows an OPEN operation or a BOF condition results in an error condition. The error indicator in positions 73 and 74, if specified, is set on or the 'E' extender, checked with %ERROR, if specified, is set on. The file *must* be repositioned using a CHAIN, SETLL, READ, READE or READP with *search-arg* specified, prior to issuing a READPE operation with factor 1 blank. A SETGT operation code should not be used to position the file prior to issuing a READPE (with no *search-arg* specified) as this results in a record-not-found condition (because the record previous to the current record never has the same key as the current record after a SETGT is issued). If *search-arg* is specified with the same key for both operation codes, then this error condition will not occur.

For more information, see "File Operations" on page 363.

Note: Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* With Factor 1 Specified...
C*
C* The previous record is read and the key compared to FieldA.
C* Indicator 99 is set on if the record's key does not match
C* FieldA.
C     FieldA        READPE    FileA                                  99
C*
C* The previous record from record format RecA is read, and
C* the key compared to FieldC.  Indicator 88 is set on if the
C* operation is not completed successfully, and 99 is set on if
C* the record key does not match FieldC.
C     FieldC        READPE    RecA                                 8899
C*
C* With No Factor 1 Specified...
C*
C* The previous record in the access path is retrieved if its
C* key value equals the key value of the current record.
C* Indicator 99 is set on if the key values are not equal.
C                   READPE    FileA                                  99
C*
C* The previous record from record format RecA is retrieved if
C* its key value matches the key value of the current record in
C* the access path.  Indicator 88 is set on if the operation is
C* not successful; 99 is set on if the key values are unequal.
C                   READPE    RecA                                 8899
```

Figure 310. READPE Operation

READS (Read Selected)

Free-Form Syntax	READS{(E)} *subfile-name* {*subfile-index*}

Code	Factor 1	Factor 2	Result Field	Indicators		
READS (E)		subfile-name	subfile-index	_	ER	EOF

The READS operation retrieves records selected from a subfile part. The first record selected from the subfile part is read.

If the subfile's selection style is extended or multiple, the record is deselected. If the subfile's selection style is single, the record remains selected. A subsequent READS reads the same record again.

The *subfile-name* operand specifies the name of the subfile part.

If the *subfile-index* operand is specified, it must be a numeric field with no decimal positions. The subfile index number of the record retrieved is placed in the *subfile-index* operand.

To handle READS exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that will be set on if an EOF condition occurs: that is, when there are no selected records in the subfile. This information can also be obtained from the %EOF built-in function, which returns '1' if an EOF condition occurs and '0' otherwise.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* SUBCUST is a subfile part which displays a list of records.
C* The READS operation reads the records selected in the displayed
C* subfile one by one in the do while loop. SCUSNO and SCUSNAM
C* are fields defined in the subfile.
C*
C                   READS     SUBCUST                                27
C                   DOW       *IN27 = *OFF
C*
C* Fields SCUSNO, SCUSNAM can be used here to process the selected
C* record which has been read.
C*
C                   READS     SUBCUST                                27
C                   ENDDO
C*
```

Figure 311. READS Operation

REALLOC (Reallocate Storage with New Length)

Free-Form Syntax	(not allowed - use the %REALLOC built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
REALLOC (E)		Length	Pointer	_	ER	_

The REALLOC operation changes the length of the heap storage pointed to by the result-field pointer to the length specified in factor 2. The result field of REALLOC contains a basing pointer variable. The result field pointer must contain the value previously set by a heap-storage allocation operation (either an ALLOC or REALLOC operation in RPG, or some other heap-storage function). It is not sufficient to simply point to heap storage; the pointer must be set to the beginning of an allocation.

New storage is allocated of the specified size and the value of the old storage is copied to the new storage. Then the old storage is deallocated. If the new length is shorter, the value is truncated on the right. If the new length is longer, the new storage to the right of the copied data is uninitialized.

The result field pointer is set to point to the new storage.

If the operation does not succeed, an error condition occurs, but the result field pointer will not be changed. If the original pointer was valid and the operation failed because there was insufficient new storage available(status 425), the original storage is not deallocated, so the result field pointer is still valid with its original value.

If the pointer is valid but it does not point to storage that can be deallocated, then status 00426 (error in storage management operation) will be set.

To handle exceptions with program status codes 425 or 426, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Factor 2 contains a numeric variable or constant that indicates the new size of the storage (in bytes) to be allocated. Factor 2 must be numeric with zero decimal positions. The value must be between 1 and 16776704.

```
D Ptr1            S               *
D Fld             S          32767A     BASED(Ptr1)
D*
C* The ALLOC operation allocates 7 bytes to the pointer Ptr1.
C* After the ALLOC operation, only the first 7 bytes of variable
** Fld can be used.
C                   ALLOC     7                Ptr1
C                   EVAL      %SUBST(Fld : 1 : 7) = '1234567'
C*
C                   REALLOC   10               Ptr1
C* Now 10 bytes of Fld can be used.
C                   EVAL      %SUBST(Fld : 1 : 10) = '123456789A'
```

Figure 312. REALLOC Operation

For more information, see "Memory Management Operations" on page 367.

RESET (Reset)

Free-Form Syntax	RESET{(E)} {*NOKEY} {*ALL} *name*

Code	Factor 1	Factor 2	Result Field	Indicators		
RESET (E)		*ALL	Variable	_	ER	_
RESET (E)	*NOKEY	*ALL	Structure	_	ER	_
RESET (E)			Window or Subfile	_	ER	_

The RESET operation sets the following to their initial value:
- Elements in a structure (record format, data structure, array, table)
- Variables (field, subfield, indicator)
- Static text and entry field parts on a window

To handle RESET exceptions (program status code 123), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

The RESET operation increases the amount of storage required by the program since any variable, structure or window that is reset the storage is doubled. For multiple occurrence data structures, tables and arrays, the initial value of every occurrence or element is saved.

Do not use the RESET during the initialization routine. If an operation (such as GOTO) is used to leave the initialization subroutine prior to where the initial values are saved, an error occurs for all RESET operations that are attempted in the program.

For more information, see "Initialization Operations" on page 366.

Resetting Entry Fields and Static Text on a Window

If the result field is a window name, factor 1 and factor 2 must be blank.

When a window is reset, entry field parts and static text parts on the window are reset to their initial values. The parts are reset to the initial values of the corresponding program fields; they are not reset to the initial values provided while using the GUI designer. The initial values of the corresponding fields is the value they had at the end of the program initialization. This value is set using the GUI designer, on the definition specification, or using the initialization subroutine. Values provided in the initialization subroutine (*INZSR) override those provided on a definition specification, and those on a definition specification override those provided to the GUI designer.

For example, the following table shows how values for various entry field parts provided in the GUI designer, and values for fields on the definition specification and in the initialization subroutine (*INZSR) affect the RESET operation:

GUI designer	Definition specification	*INZSR	Value after RESET
ENT0000B=22.5	ENT0000B=30.5		ENT0000B=30.5
ENT0000A=abc	ENT0000A=xyz	ENT0000A=pqr	ENT0000A=pqr

GUI designer	Definition specification	*INZSR	Value after RESET
		ENT0000C=Name	ENT0000C=Name
			If ENT0000D is character, RESET resets to blanks. If ENT0000D is numeric, RESET resets to zero.

Note: After the RESET operation, the values stored in the program fields match the values seen on the display.

Resetting Elements in a Structure and Variables

The initial values for a variable or structure is the value they had at the end of the program initialization. This value can be set using the INZ keyword on the definition specification or using the initialization subroutine to assign an initial value. This initial value is used by the RESET operation. Values provided in the initialization subroutine (*INZSR) override those provided on a definition specification.

The result field must contain a record format name, data structure name, array name, table name, field name, subfield, array element, or indicator name:

- If a record format or a single occurrence data structure is being reset, all fields are reset in the order they are declared within the structure.

  If factor 1 is specified, it must contain *NOKEY which indicates that the key fields are not reset to their initial values.

  If factor 2 is specified, it must contain *ALL which indicates that all fields for the record format are reset. If factor 2 is not specified, only output fields in the record format are affected. All field conditioning indicators of the record format are affected. Input-only fields are not affected by RESET.
- If a multiple occurrence data structure is being reset, all fields in the current occurrence are reset.
- If an array is being reset, the entire array is reset.
- If a table is being reset, the current table element is reset.
- If an array element (including indicators) is being reset, only the element specified is reset.

Note: RESET is not allowed for based variables.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FEXTFILE   O    E             DISK    REMOTE
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D
D* The file EXTFILE contains one record format RECFMT containing
D* the character fields CHAR1 and CHAR2 and the numeric fields
D* NUM1 and NUM2.
D
D DS1             DS
D  DAY1                   1      8    INZ('MONDAY')
D  DAY2                   9     16    INZ('THURSDAY')
D  JDATE                 17     22
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following operation sets DAY1, DAY2, and JDATE to blanks.
C
C                   CLEAR                   DS1
C
C* The following operation will set DAY1, DAY2, and JDATE to their
C* initial values of 'MONDAY', 'THURSDAY', and UDATE respectively.
C* The initial value of UDATE for JDATE is set in the *INZSR.
C
C                   RESET                   DS1
C
C* The following operation will set CHAR1 and CHAR2 to blanks and
C* NUM1 and NUM2 to zero.
C                   CLEAR                   RECFMT
C* The following operation will set CHAR1, CHAR2, NUM1, and
C* NUM2 to their initial values of 'NAME', 'ADDRESS', 1, and 2
C* respectively. These initial values are set in the *INZSR.
C*
C                   RESET                   RECFMT
C
C     *INZSR        BEGSR
C                   MOVEL     UDATE         JDATE
C                   MOVEL     'NAME    '    CHAR1
C                   MOVEL     'ADDRESS '    CHAR2
C                   Z-ADD     1             NUM1
C                   Z-ADD     2             NUM2
C                   ENDSR
C* The following operation sets all fields in the record format
C* to blanks, except the key fields.
C*
C     *NOKEY        RESET    *ALL           DBRECFMT
```

Figure 313. RESET Operation

RETURN (Return to Caller)

Free-Form Syntax	RETURN{(HMR)} *expression*

Code	Factor 1	Extended Factor 2
RETURN (H M/R)		expression

The RETURN operation causes a return to the caller:

- If LR is on, the program ends normally and the component is terminated. *TERMSR is performed. Any locked data area structures, arrays, and tables are written. External indicators are reset. If more than on subroutine has been invoked, RETURN causes a return to the previous action subroutine invocation.
- If LR is not on, default processing associated with the current event is performed, unless the RETURN is in a nested subroutine, or initialization or termination is being performed.

Note: LR has no effect until the last action subroutine invocation returns.

When a subprocedure returns, the return value, if specified on the prototype of the called program or procedure, is passed to the caller. Nothing else occurs automatically. All files and data areas must be closed manually. You can set on indicators but this will not cause program termination to occur. For information on how operation extenders H, M, and R are used, see “Precision Rules for Numeric Operations” on page 390.

In a subprocedure that returns a value, a RETURN operation must be coded within the subprocedure. The actual returned value has the same role as the left-hand side of the EVAL expression, while the *expression* operand of the RETURN operation has the same role as the right-hand side. An array may be returned only if the prototype has defined the return value as an array.

In a subprocedure that returns a value, you should ensure that a RETURN operation is performed before reaching the end of the procedure. If the subprocedure ends without encountering a RETURN operation, an exception is signalled to the caller.

For more information, see “Call Operations” on page 353.

ROLBK (Roll Back)

Free-Form Syntax	ROLBK{(E)}

Code	Factor 1	Factor 2	Result Field	Indicators		
ROLBK (E)				_	ER	_

The ROLBK operation eliminates all changes to any OS/400 database files that have been opened for commitment control since the previous commit or rollback operations or since the beginning of operations under commitment control if there has been no previous COMMIT or ROLBK.

The ROLBK operation can only be used with OS/400 files. It cannot be used with local files.

All record locks for files under commitment control for a particular server are released regardless of which component issued the ROLBK.

Note: The component issuing the ROLBK does not need to have any file under commitment control.

To handle ROLBK exceptions (program status codes 802 to 805), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "File Operations" on page 363.

SCAN (Scan String)

Free-Form Syntax	(not allowed - use the %SCAN built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
SCAN (E)	Compare string:length	Base string:start	Left-most position	_	ER	FD

The SCAN operation scans a base string for a compare string. The SCAN begins at the leftmost character of the base string in factor 2 (as specified by the start location) and continues character by character, from left to right, comparing the characters in factor 2 to those in factor 1. The strings are indexed from position 1.

Notes:

1. The compare string and base string must both be of the same type, either character, graphic, or UCS-2.
2. Leading, trailing, or embedded blanks specified in the compare string are included in the SCAN operation.
3. The SCAN operation is case-sensitive. A compare string specified in lowercase will not be found in a base string specified in uppercase.
4. Figurative constants cannot be used in the factor 1, factor 2, or result fields.
5. No overlapping within data structures is allowed for factor 1 and the result field or factor 2 and the result field.

Factor 1 must contain either the compare string or the compare string, followed by a colon, followed by the length. The compare string must contain a field name, array element, named constant, data structure name, literal, or table name. The length must be numeric with no decimal positions must contain a named constant, array element, field name, literal, or table name. If no length is specified, the compare string is used.

Factor 2 must contain either the base string or the base string, followed by a colon, followed by the start location. The base string must contain a field name, array element, named constant, data structure name, literal, or table name. The start location must be numeric with no decimal positions and must be a named constant, array element, field name, literal, or table name. If no start location is specified, the default value is 1.

Note: The start cannot be greater than the length.

If graphic or UCS-2 strings are used, the start position and length are measured in double bytes.

If the start position is greater than 1, the result field contains the position of the compare string relative to the beginning of the source string, not relative to the start position.

The result field contains the value of the leftmost position of the compare string in the base string, if found. It must be numeric with no decimal positions and must contain a field name, array element, array name, or table name. If no result field is specified, a resulting indicator in positions 75 and 76 must be specified. The result field is set to 0 if the string is not found.

If the result field contains an array, each occurrence of the compare string is placed in the array with the leftmost occurrence in element 1. The array elements following the element containing the rightmost occurrence are all zero. The result array should be as large as the field length of the base string specified in factor 2.

If the result field is a numeric array, as many occurrences as there are elements in the array are noted. If no occurrences are found, the result field is set to zero

To handle SCAN exceptions (program status code 100), either the operation code extender 'E' or an error indicator ER can be specified, but not both. An error occurs if the start position is greater than the length of factor 2 or if the value of factor 1 is too large. For more information on error handling, see "Program Exception and Errors" on page 51.

You can specify an indicator in positions 75-76 that is set on if the string being scanned for is found. This information can also be obtained from the %FOUND built-in function, which returns '1' if a match is found.

For more information, see "String Operations" on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The SCAN operation finds the substring 'ABC' starting in
C* position 3 in factor 2;  3 is placed in the result field.
C* Indicator 90 is set on because the string is found.  Because
C* no starting position is specified, the default of 1 is used.
C     'ABC'         SCAN      'XCABCD'      RESULT                  90
C*
C* This SCAN operation scans the string in factor 2 for an
C* occurrence of the string in factor 1 starting at position 3.
C* The 'Y' in position 1 of the base string is ignored because
C* the scan operation starts from position 3.
C* The operation places the values 5 and 6 in the first and
C* second elements of the array.  Indicator 90 is set on.
C
C                   MOVE      'YARRYY'      FIELD1            6
C                   MOVE      'Y'           FIELD2            1
C     FIELD2        SCAN      FIELD1:3      ARRAY                   90
C
C* This SCAN operation scans the string in factor 2, starting
C* at position 2, for an occurrence of the string in factor 1
C* for a length of 4.  Because 'TOOL' is not found in FIELD1,
C* INT is set to zero and indicator 90 is set off.
C
C                   MOVE      'TESTING'     FIELD1            7
C                   Z-ADD     2             X                 1 0
C                   MOVEL     'TOOL'        FIELD2            5
C     FIELD2:4      SCAN      FIELD1:X      INT90            20      90
C
C*
C* This SCAN operation is searching for a name.  When the name
C* is found, %FOUND returns '1' so HandleLine is called.
C*
C     SrchName      SCAN      Line
C                   IF        %FOUND
C                   EXSR      HandleLine
C                   ENDIF
```

Figure 314. SCAN Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
D*
D*       A Graphic SCAN example
D*
D*       Value of Graffld is graphic 'AACCBBGG'.
D*       Value of Number after the scan is 3 as the 3rd graphic
D*       character matches the value in factor 1
D
D Graffld                         4G   inz(G'AACCBBGG')
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C* The SCAN operation scans the graphic string in factor 2 for
C* an occurrence of the graphic literal in factor 1. As this is a
C* graphic operation, the SCAN will operate on 2 bytes at a time
C
C     G'BB'         SCAN      Graffld:2     Number            5 0    90
C
```

Figure 315. SCAN Operation using graphic

SELECT (Begin a Select Group)

Free-Form Syntax	SELECT

Code	Factor 1	Factor 2	Result Field	Indicators		
SELECT						

The select group conditionally processes one of several alternative sequences of operations. It consists of:
- A SELECT statement
- Zero or more WHENxx or WHEN groups
- An optional OTHER
- ENDSL or END statement.

After the SELECT operation, control passes to the statement following the first WHENxx condition that is satisfied. All statements are then executed until the next WHENxx operation. Control passes to the ENDSL statement (only one WHENxx is executed). If no WHENxx condition is satisfied and an OTHER operation is specified, control passes to the statement following the OTHER operation. If no WHENxx condition is satisfied and no OTHER operation is specified, control transfers to the statement following the ENDSL operation of the select group.

Conditioning indicators can be used on the SELECT operation. If they are not satisfied, control passes immediately to the statement following the ENDSL operation of the select group. Conditioning indicators cannot be used on WHENxx, WHEN, OTHER and ENDSL operation individually.

The select group can be nested within IF, DO, or other select groups. The IF and DO groups can be nested within select groups.

If a SELECT operation is specified inside a select group, the WHENxx and OTHER operations apply to the new select group until an ENDSL is specified.

For more information, see “Structured Programming Operations” on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* In the following example, if X equals 1, do the operations in
C* sequence 1 (note that no END operation is needed before the
C* next WHENxx); if X does NOT equal 1, and if Y=2 and X<10, do the
C* operations in sequence 2.  If neither condition is true, do
C* the operations in sequence 3.
C*
C                   SELECT
C                   WHEN      X = 1
C                   Z-ADD     A             B
C                   MOVE      C             D
C*  Sequence 1
C                   :
C                   WHEN      ((Y = 2) AND (X < 10))
C*  Sequence 2
C                   :
C                   OTHER
C*  Sequence 3
C                   :
C                   ENDSL
C*
C* The following example shows a select group with conditioning
C* indicators.  After the CHAIN operation, if indicator 10 is on,
C* then control passes to the ADD operation.  If indicator 10 is
C* off, then the select group is processed.
C*
C     KEY           CHAIN     FILE                              10
C  N10              SELECT
C                   WHEN      X = 1
C*  Sequence 1
C                   :
C                   WHEN      Y = 2
C*  Sequence 2
C                   :
C                   ENDSL
C                   ADD       1              N
```

Figure 316. SELECT Operation

SETATR (Set Attribute)

Free-Form Syntax	(not allowed - use the %SETATR built-in function or “Qualified GUI Part Attribute Access” on page 379)

Code	Factor 1	Factor 2	Result Field	Indicators		
SETATR (E)	part name	attribute value	attribute	_	ER	_

The SETATR operation sets the attribute of a part. A part’s attribute can be set only if that part has been created.

Notes:

1. The SETATR operations can be used for multiple link action subroutines. For a description of multiple link action subroutines, see “BEGACT (Begin Action Subroutine)” on page 508. To set an attribute for a part on a window other than the parent window, use the %SETATR built-in function. For a description of the %SETATR built-in function, see “%SETATR (Set Attribute)” on page 471.
2. The SETATR operation does not support 1-byte and 8-byte signed and unsigned integer values, and unicode values.

If factor 1 is specified, it must contain the name of the part or a field containing the name of a part whose attribute is being set.

Factor 2 must contain the new value for the attribute. It must be a literal, named constant, figurative constant, or a field containing the new value for the attribute.

The result field must contain the attribute name or a field containing the name of the attribute.

To handle SETATR exceptions (program status codes1400, 1402-1404, either the operation code extender ’E’ or an error indicator ER can be specified, but not both. For more information on error handling, see “Program Exception and Errors” on page 51.

If a resulting indicator is specified, it is set on when the SETATR operation does not complete successfully.

Note: The %SETATR built-in function does not affect the corresponding program fields for parts. To ensure that the attribute value and the value in the program field are the same, use the program field when setting the attribute value. This applies to attributes that have program fields mapped to them, such as entry fields with the TEXT attribute.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C* Change the label on a button called BUTTON1.
C*
C     'BUTTON1'     SETATR    Cancel        'LABEL'
```

Figure 317. SETATR Operation

SETGT (Set Greater Than)

Free-Form Syntax	SETGT{(EHMR)} *search-arg name*

Code	Factor 1	Factor 2	Result Field	Indicators		
SETGT (E)	search-arg	name (file or record format)		NR	ER	_

The SETGT operation positions a file at the next record with a key or relative record number that is greater than the search argument. The file must be a full procedural file (identified by an F in position 18 of the file description specifications).

The SETGT operation can only be used with OS/400 files.

The search argument, *search-arg*, must be the key or relative record number used to retrieve the record. If access is by key, *search-arg* can be a a single key in the form of a field name, a named constant, a figurative constant, or a literal. See Figure 227 on page 531 for an example of searching key fields.

If the file is an externally-described file, *search-arg* can also be a composite key in the form of a KLIST name, a list of values, or %KDS. Graphic and UCS-2 key fields must have the same CCSID as the key in the file. See the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451 for an illustration of search arguments in a data structure. If access is by relative record number, *search-arg* must be an integer literal or a numeric field with zero decimal positions.

The *name* operand is required and must be either a file name or a record format name. A record format name is allowed only with an externally described file. If MBR(*ALL) is specified, SETGT only processes the first open file member.

You can specify an indicator in positions 71-72 that is set on if no record is found with a key or relative record number that is greater than the search argument specified (*search-arg*). This information can also be obtained from the %FOUND built-in function, which returns '0' if no record is found, and '1' if a record is found.

To handle SETGT exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

If the SETGT operation is not successful (no-record-found condition), the file is positioned to the end of the file.

Once the SETGT operation is performed, the file is positioned so that it is immediately before the first record whose key or relative record number is greater than the search argument specified (*search-arg*). This record can be retrieved reading the file. You can use *START and *END for file positioning. If you specify either *START or *END for *search-arg*, note the following:

- *name* must be a file name.
- You cannot use *HIVAL or *LOVAL as the *search-arg*.
- You cannot specify the NR indicator.

For more information, see "File Positioning" on page 6.

See "Database Null Value Support" on page 137 for information on reading records with null-capable fields.

For more information, see "File Operations" on page 363.

Note: Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*  This example shows how to position the file so READ will read
C*  the next record. The search argument, KEY, specified for the
C*  SETGT operation has a value of 98; therefore, SETGT positions
C*  the file before the first record of file format FILEA that
C*  has a key field value greater than 98.  The file is positioned
C*  before the first record with a key value of 100.  The READ
C*  operation reads the record that has a value of 100 in its key
C*  field.
C
C     KEY           SETGT     FILEA
C                   READ      FILEA                                64
C*
C*  This example shows how to read the last record of a group of
C*  records with the same key value and format from a program
C*  described file.  The search argument, KEY, specified for the
C*  SETGT operation positions the file before the first record of
C*  file FILEB that has a key field value greater than 70.
C*  The file is positioned before the first record with a key
C*  value  of 80.  The READP operation reads the last record that
C*  has a value of 70 in its key field.
C
C     KEY           SETGT     FILEB
C                   READP     FILEB                                64
```

Figure 318. SETGT Operation

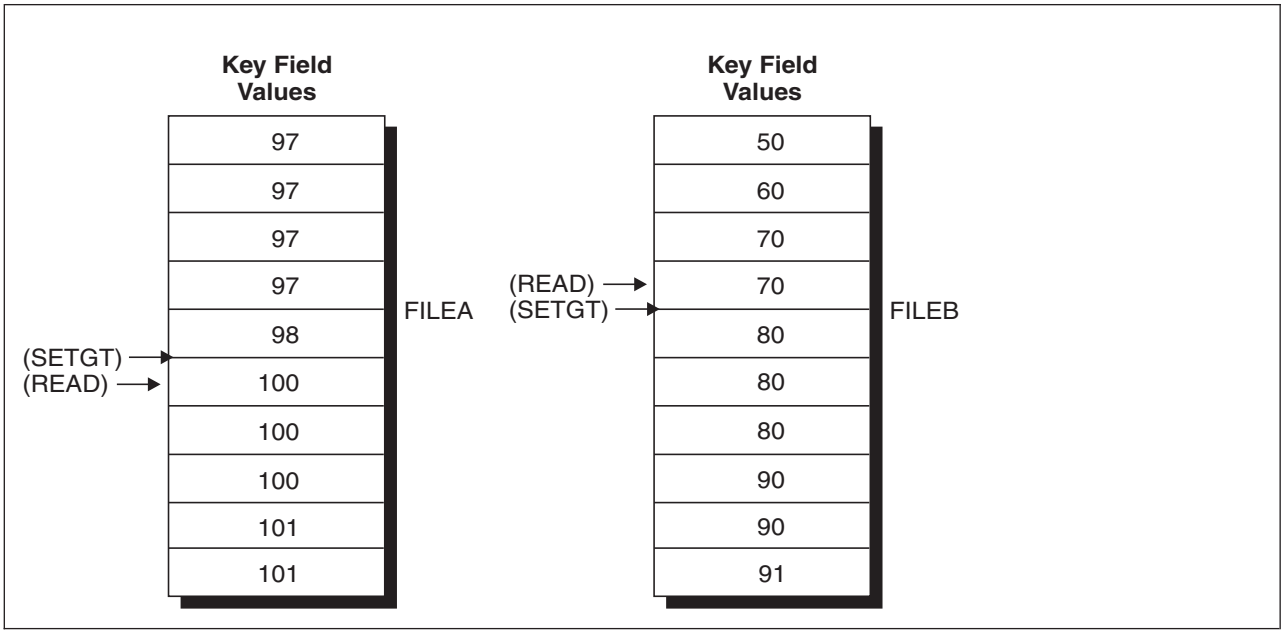


Figure 319. Positioning files using SETGT

SETLL (Set Lower Limit)

Free-Form Syntax	SETLL{(EHMR)} *search-arg name*

Code	Factor 1	Factor 2	Result Field	Indicators		
SETLL (E)	search-arg	name (file or record format)		NR	ER	EQ

The SETLL operation positions a file at the next record that has a key or relative record number that is greater than or equal to the search argument. The file must be a full procedural file (identified by an F in position 18 of the file description specifications).

The SETLL operation can only be used with OS/400 files. It cannot be used with local files.

The search argument, *search-arg*, must be the key or relative record number used to retrieve the record. If access is by key, *search-arg* can be a a single key in the form of a field name, a named constant, a figurative constant, or a literal. See Figure 227 on page 531 for an example of searching key fields.

If the file is an externally-described file, *search-arg* can also be a composite key in the form of a KLIST name, a list of values, or %KDS. Graphic and UCS-2 key fields must have the same CCSID as the key in the file. See the example at the end of "%KDS (Search Arguments in Data Structure)" on page 451 for an illustration of search arguments in a data structure. If access is by relative record number, *search-arg* must be an integer literal or a numeric field with zero decimal positions.

The *name* operand is required and can contain either a file name or a record format name. A record format name is allowed only with an externally described file. If MBR(*ALL) is specified, SETLL only processes the first open file member.

The resulting indicators reflect the status of the operation. You can specify an indicator in positions 71-72 that is set on when the search argument is greater than the highest key or relative record number in the file. This information can also be obtained from the %FOUND built-in function, which returns '0' if no record is found, and '1' if a record is found.

To handle SETLL exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 that is set on when a record is present whose key or relative record number is equal to the search argument. This information can also be obtained from the %EQUAL built-in function, which returns '1' if an exact match is found.

If *name* is a file name for which the lower limit is to be set, the file is positioned at the first record with a key or relative record number equal to or greater than the search argument.

If *name* is a record format name for which the lower limit is to be set, the file is positioned at the first record of the specified type that has a key equal to or greater than the search argument.

When end of file is reached on a file being processed by SETLL, another SETLL can be issued to reposition the file. After a SETLL operation successfully positions the file at a record, the record can be retrieved by reading it. You can use *START and *END for file positioning. If you specify either *START or *END for *search-arg*, note the following:

- *name* must be a file name.
- You cannot use *HIVAL or *LOVAL for *search-arg*.
- You cannot specify the NR or EQ indicator.
- Either an error indicator (positions 73-74) or the 'E' extender may be specified.

For more information on using *START and *END, see "File Positioning" on page 6.

Before your application reads this file, another application may delete records from the file. You may not retrieve the record that you expect. Even if the %EQUAL built-in function is also set on or the resulting indicator in positions 75 and 76 is set on to indicate that a matching record has been found, you may not get that record.

SETLL does not access data records. To verify that a key exists, use SETLL with an equal indicator (positions 75-76) or the %EQUAL built-in function rather than the CHAIN operation. In cases where the file is a multiple format logical file with sparse keys, CHAIN can be a faster solution than SETLL.

See "Database Null Value Support" on page 137 for information on reading records with null-capable fields.

For more information, see "File Operations" on page 363.

Note: Operation code extenders H, M, and R are allowed only when the search argument is a list or is %KDS().

In the following example, the file ORDFIL contains order records. The key field is the order number (ORDER) field. There are multiple records for each order. ORDFIL looks like this in the calculation specifications:

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  All the 101 records in ORDFIL are to be printed.  The value 101
C*  has previously been placed in ORDER.  The SETLL operation
C*  positions the file at the first record with the key value 101
C*  %EQUAL will return '1'.
C
C     ORDER         SETLL     ORDFIL
C
C*  The following DO loop processes all the records that have the
C*  same key value.
C*
C                   IF        %EQUAL
C                   DOU       %EOF
C     ORDER         READE     ORDFIL
C                   IF        NOT %EOF
C                   EXCEPT    DETAIL
C                   ENDIF
C                   ENDDO
C                   ENDIF
C
C*
C*  The READE operation reads the second, third, and fourth 101
C*  records in the same manner as the first 101 record was read.
C*  After the fourth 101 record is read, the READE operation is
C*  attempted.  Because the 102 record is not of the same group,
C*  %EOF will return '1', the EXCEPT operation is bypassed, and
C*  the DOU loop ends.
```

Figure 320. SETLL Operation

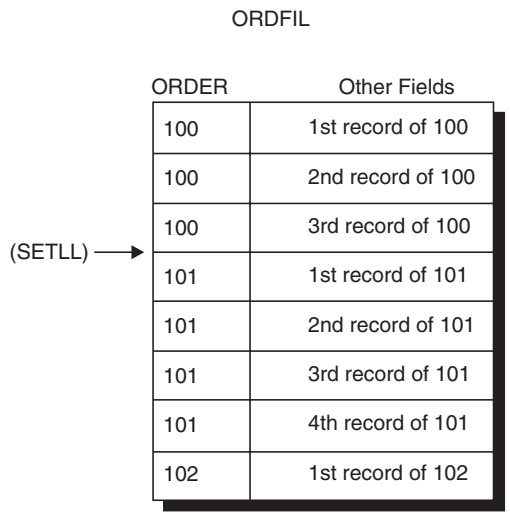


Figure 321. Positioning files using SETLL

SETOFF (Set Indicator Off)

Free-Form Syntax	(not allowed - use EVAL *INxx = *OFF)

Code	Factor 1	Factor 2	Result Field	Indicators		
SETOFF				OF	OF	OF

The SETOFF operation sets off any indicators specified in positions 71 through 76. You must specify at least one resulting indicator in positions 71 through 76.

Figure 322 illustrates the SETOFF operation.

For more information, see "Indicator-Setting Operations" on page 366.

SETON (Set Indicator On)

Free-Form Syntax	(not allowed - use EVAL *INxx = *ON)

Code	Factor 1	Factor 2	Result Field	Indicators		
SETON				ON	ON	ON

The SETON operation sets on any indicators specified in positions 71 through 76. You must specify at least one resulting indicator in positions 71 through 76.

For more information, see "Indicator-Setting Operations" on page 366.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The SETON and SETOFF operations set from one to three indicators
C*  specified in positions 71 through 76 on and off.
C*  The SETON operation sets indicator 17 on.
C
C                   SETON                                        17
C
C*  The SETON operation sets indicators 17 and 18 on.
C
C                   SETON                                        1718
C
C*  The SETOFF operation sets indicator 21 off.
C
C                   SETOFF                                       21
```

Figure 322. SETON and SETOFF Operations

SHOWWIN (Display Window)

Free-Form Syntax	SHOWWIN{(E)} *window-name*}

Code	Factor 1	Factor 2	Result Field	Indicators		
SHOWWIN (E)		Window name		_	ER	_

The SHOWWIN operation loads a window into memory. The Visible attribute controls whether the window is displayed.

Note: The attributes for a window cannot be set or referenced before a SHOWWIN operation. Parts on a window cannot be referenced before a SHOWWIN operation for the window.

Factor 2 contains the name of the window to be displayed.

To handle SHOWWIN exceptions (program status codes 1400 to 1420), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Do not use SHOWWIN and the Open Immediately attribute. If you do, the resulting indicator is set on. This means that the window is already loaded. If this indicator is set on, the Visible attribute for the window can be set.

Use the SHOWWIN operation to display windows that are not displayed frequently rather than setting the window to Open Immediately. For primary windows (the first window an application displays), use the Open Immediately setting for the window rather than SHOWWIN.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C* A window named UPDCUST is displayed.
C                    SHOWWIN   'UPDCUST'
```

Figure 323. SHOWWIN Operation

SORTA (Sort an Array)

Free-Form Syntax	SORTA *array-name*
	SORTA %SUBARR(array-name : start-element { : number-of-elements })

Code	Factor 1	Extended Factor 2
SORTA		Array
SORTA		%SUBARR(Array : start-element {:number-of-elements})

The SORTA operation sorts the array specified by *array-name* into either an ascending or descending sequence. This sequence is specified on the definition specification. If no sequence is specified, the array is sorted into ascending sequence.

The *array-name* operand is the name of an array to be sorted. *IN cannot be specified. If the array is defined as a compile-time or pre-runtime array with data in alternating form, the alternate array is not sorted.

If the array is defined with the OVERLAY(name{:pos}) keyword, the base array will be sorted in the sequence defined by the OVERLAY array.

Graphic arrays are sorted by the hexadecimal values of the array elements, in the order specified on the definition specification.

To sort a portion of an array, use the %SUBARR built-in function.

Sorting an array does not preserve any previous order. For example, if you sort an array twice, using different overlay arrays, the final sequence is that of the last sort. Elements that are equal in the sort sequence but have different hexadecimal values (for example, due to the use of an overlay array to determine sequence), may not be in the same order after sorting as they were before.

When sorting arrays of basing pointers, you must ensure that all values in the arrays are addresses within the same space. Otherwise, inconsistent results may occur. See “Compare Operations” on page 357 for more information.

If a null-capable array is sorted, the sorting will not take the settings of the null flags into consideration.

Sorting a dynamically allocated array without all defined elements allocated may cause errors to occur.

For more information, see “Array Operations” on page 351.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
DARRY             S              1A   DIM(8) ASCEND
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The SORTA operation sorts ARRY into ascending sequence because
C*  the ASCEND keyword is specified.
C*  If the unsorted ARRY contents were GT1BA2L0, the sorted ARRY
C*  contents would be 012ABGLT.
C*  Note that the ASCII sorting sequence is used.
C
C                   SORTA     ARRY
```

Figure 324. SORTA Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
D* In this example, the base array has the values aa44 bb33 cc22 dd11
D* so the overlaid array ARRO has the values 44 33 22 11.
D                 DS
D ARR                             4    DIM(4) ASCEND
D ARRO                            2    OVERLAY(ARR:3)
D
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C* After the SORTA operation, the base array has the values
C* 11dd 22cc 33bb 44aa
C
C                   SORTA     ARRO
```

Figure 325. SORTA Operation with OVERLAY

SQRT (Square Root)

Free-Form Syntax	(not allowed - use the %SQRT built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
SQRT (H)		Value	Root			

The SQRT operation derives the square root of the field named in factor 2. The square root of factor 2 is placed in the result field.

Factor 2 must be numeric and can contain an array, array element, field, figurative constant, literal, named constant, subfield, or table name. If the value of the factor 2 field is zero, the result field value is also zero. If the value of the factor 2 field is negative, the VRPG Client exception/error handling routine receives control.

The result field must be numeric and can contain an array, array element, subfield, or table element.

An entire array can be used in a SQRT operation if factor 2 and the result field contain array names. The number of decimal positions in the result field can be either less than or greater than the number of decimal positions in factor 2. However, the result field should not have fewer than half the number of decimal positions in factor 2.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the SQRT operation.

START (Start Component or Call Local Program)

Free-Form Syntax	START{(E)} *name*}

Code	Factor 1	Factor 2	Result Field	Indicators		
START (E)		Component name or field name	PLIST name	_	ER	_

The START operation can be used to either start a new component in the application or to call a local program. For every START operation that starts a new component, there can be a STOP operation.

To handle START exceptions (program status code 1410), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Starting Components

Since START is an asynchronous operation, both the called (target) and calling (source) components as well as any other active components in the application can receive events for parts currently enabled by all the application's components.

If factor 2 is specified, it must contain a character literal which is the name of the component or a variable containing the component name.

If the result field is specified, it must contain a PLIST name. If the result field is not specified, the START operation can be followed by a PARM operation. Parameters are passed by address. This means that the source and target components can access parameter fields. Up to 20 parameters can be specified. If the started component is expecting a varying length field, then a varying length field must be used as the parameter.

START initializes a component, executes its *INZSR, then returns to the source component. In the source component, factor 2 of the PARM operation is copied to the result field of the PARM operation. When control returns to the source component, the result field is copied to factor 1. In the target component, the result field is copied to factor 1. When control returns to the source component, factor 2 is copied to the result field if the *INZSR is successful.

Once the START operation has finished initializing the target component, the action subroutine in the source component continues to execute and the target component remains active with its action subroutines enabled to receive events.

Calling Local Programs

If START is used to call a local program, the calling program makes the call to the local program, then continues. The called program runs independently of the calling program.

If factor 2 is specified, it must contain a definition specification name for a constant or a field definition. LINKAGE(*CLIENT) must be on the definition specification. For more information, see “LINKAGE(linkage_type)” on page 281.

If the result field is specified, it must contain a PLIST name. If the result field is not specified, the START operation can be followed by a PARM operation. Parameters are passed by reference. See “PARM (Identify Parameters)” on page 647 and “PLIST (Identify a Parameter List)” on page 650. for more information on passing parameters.

Note: Pointers and procedure pointers are not allowed as parameters.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++Comments++++++++++++
Dtest2            C                   'testprog'   LINKAGE (*CLIENT)
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                   START     test2
```

Figure 326. START Operation

STOP (Stop Component)

Free-Form Syntax	STOP{(E)} *component-name*}

Code	Factor 1	Factor 2	Result Field	Indicators		
STOP (E)		Component name			ER	

The STOP operation stops one or more components in an application. Any child components that may have been started by the component being terminated are terminated first.

When a component terminates:
1. Components that have been started by the terminating component or the component's descendants are terminated in reverse hierarchical order.
2. The *TERMSR subroutine is called for each component being terminated that has a *TERMSR defined.
3. *STATUS codes are placed in the PSDS.

If factor 2 is specified, it must contain a character literal which is the component name or a variable containing the component name. If factor 2 is not specified or if factor 2 contains the same component name as the component currently running (the component that contains the STOP operation), then that component terminates.

When a STOP is performed which affects a currently running component, operations following the STOP are not executed. For example, COMPA starts COMPB. If COMPB is the component that is currently executing and if COMPB issues a STOP for COMPA, COMPB terminates first, then COMPA terminates. No operations following the STOP are performed.

To handle STOP exceptions, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                            Extended-factor2++++++++++++++++++++++++++++
C*
C                   STOP      'COMPX'
```

Figure 327. STOP Operation

SUB (Subtract)

Free-Form Syntax	(not allowed - use the - or -= operators)

Code	Factor 1	Factor 2	Result Field	Indicators		
SUB (H)	Minuend	Subtrahend	Difference	+	–	Z

If factor 1 is specified, factor 2 is subtracted from factor 1 and the difference is placed in the result field. If factor 1 is not specified, factor 2 is subtracted from the result field.

Factor 1 and factor 2 must be numeric, and each can an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric, and can contain an array, array element, subfield, or table name.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the SUB operation.

SUBDUR (Subtract Duration)

Free-Form Syntax	not allowed - use the - or -= operators with duration functions such as %YEARS and %MONTHS, or the %DIFF built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
SUBDUR (E) (duration)	Date/Time/ Timestamp	Date/Time/ Timestamp	Duration: Duration Code	_	ER	_
SUBDUR (E) (new date)	Date/Time/ Timestamp	Duration:Duration Code	Date/Time/ Timestamp	_	ER	_

The SUBDUR operation has been provided to:
- Subtract a duration to establish a new Date, Time or Timestamp
- Calculate a duration

Figure 328 on page 695 illustrates the SUBDUR operation.

Subtract a duration

You can use the SUBDUR operation to subtract a duration specified in factor 2 from a field or constant specified in factor 1 and place the resulting resulting Date, Time, or Timestamp in the field specified in the result field.

Factor 1 is optional and may contain a Date, Time or Timestamp field, array, array element, literal or constant. If factor 1 contains a field name, array or array element, then its data type must be the same type as the field specified in the result field. If factor 1 is not specified, the duration is subtracted from the field specified in the result field.

Factor 2 is required and contains two subfactors. The first is a duration which must be a numeric field, array or constant with zero decimal positions. The duration must be 15 digits or less. If the duration field is negative, then duration is added to the field.

The second subfactor must be a valid duration code indicating the type of duration. The duration code must be consistent with the result field data type. For example, you can subtract a year, month, or day duration but not a minute duration from a date field. For list of duration codes and their short forms, see “Date Operations” on page 359.

The result field must be a Date, Time or Timestamp data type field, array or array element. If factor 1 is left blank, the duration is subtracted from the value in the result field. If the result field is an array, the value in factor 2 is subtracted from each element in the array. If the result is a time field, the result will always be be a valid Time. For example, subtracting 59 minutes from 00:58:59 would give -00:00:01. Since this time is not valid, the compiler adjusts it to 23:59:59.

When subtracting a duration in months from a date, the general rule is that the month portion is decreased by the number of months in the duration, and the day portion is unchanged. The exception to this is when the resulting day portion would exceed the actual number of days in the resulting month. In this case, the resulting day portion is adjusted to the actual month end date. The following examples, which assume a *YMD format, illustrate this point.

```
'95/05/30' SUBDUR 1:*MONTH results in '95/04/30'
```

The resulting month portion has been decreased by 1; the day portion is unchanged.

```
'95/05/31' SUBDUR 1:*MONTH results in '95/04/30'
```

The resulting month portion has been decreased by 1; the resulting day portion has been adjusted because April has only 30 days.

Similar results occur when subtracting a year duration. For example, subtracting one year from '92/02/29' results in '91/02/28', an adjusted value since the resulting year is not a leap year.

To handle exceptions with program status codes 103, 112 or 113, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Calculate a duration

The SUBDUR operation can be used to calculate a duration between:

- Two dates
- A date and a timestamp
- Two times
- A time and a timestamp
- Two timestamps

Factor 1 is required and must contain a Date, Time or Timestamp field, subfield, array, array element, constant or literal.

Factor 2 is required and must also contain a Date, Time or Timestamp field, array, array element, literal or constant.

The following duration codes are valid:

- For two dates or a date and a timestamp: *DAYS (*D), *MONTHS (*M), and *YEARS (*Y)
- For two times or a time and a timestamp: *SECONDS (*S), *MINUTES (*MN), and *HOURS (*H)
- For two timestamps: *MSECONDS (*MS), *SECONDS (*S), *MINUTES (*MN), *HOURS (*H), *DAYS (*D), *MONTHS (*M), and *YEARS (*Y).

The result is a number of whole units, with any remainder discarded. For example, 61 minutes is equal to 1 hour and 59 minutes is equal to 0 hours.

The result field consists of two subfactors. The first is the name of a zero decimal numeric field, array or array element in which the result of the operation will be placed. The second subfactor contains a duration code denoting the type of duration. The result field will be negative if the date in factor 1 is earlier than the date in factor 2.

For more information on working with date-time fields see "Date Operations" on page 359.

Note: Calculating a micro-second Duration (*mseconds) can exceed the 15 digit system limit for Durations and cause errors or truncation. This situation will occur when there is more than a 32 year and 9 month difference between the factor 1 and factor 2 entries.

Possible error situations

1. For subtracting durations:
   - If the value of the Date, Time or Timestamp field in factor 1 is invalid
   - If factor 1 is blank and the value of the result field before the operation is invalid
   - or if the result of the operation is greater than *HIVAL or less than *LOVAL.
2. For calculating durations:
   - If the value of the Date, Time or Timestamp field in factor 1 or factor 2 is invalid
   - or if the result field is not large enough to hold the resulting duration.

In each of these cases an error will be signalled.

If an error is detected, an error will be generated with one of the following program status codes:
- 00103: Result field not large enough to hold result
- 00112: Date, Time or Timestamp value not valid
- 00113: A Date overflow or underflow occurred (that is, the resulting Date is greater than *HIVAL or less than *LOVAL).

The value of the result field remains unchanged. To handle exceptions with program status codes 103, 112 or 113, either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

SUBDUR Examples

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C* Determine a LOANDATE which is xx years, yy months, zz days prior to
C* the DUEDATE.
 *
C     DUEDATE       SUBDUR    XX:*YEARS     LOANDATE
C                   SUBDUR    YY:*MONTHS    LOANDATE
C                   SUBDUR    ZZ:*DAYS      LOANDATE
 *
C* Add 30 days to a loan due date
C*
C                   SUBDUR    -30:*D        LOANDUE
 *
C* Calculate the number or days between a LOANDATE and a DUEDATE.
 *
C     LOANDATE      SUBDUR    DUEDATE       NUM_DAYS:*D        5 0
 *
C* Determine the number of seconds between LOANDATE and DUEDATE.
 *
C     LOANDATE      SUBDUR    DUEDATE       NUM_SECS:*S        5 0
```

Figure 328. SUBDUR Operations

SUBST (Substring)

Free-Form Syntax	(not allowed - use %SUBST)

Code	Factor 1	Factor 2	Result Field	Indicators		
SUBST (E P)	Length to extract	Base string:start	Target string	_	ER	_

The SUBST operation returns a substring, starting at the location specified in factor 2 for the length specified in factor 1. This substring is placed in the result field. If factor 1 is not specified, the length of the string from the start position is used. For graphic or UCS-2 strings, the start position is measured in double bytes. The base string and the target string must both be of the same type, either both character, both graphic, or both UCS-2.

If factor 1 is specified, it must contain the length of the string to be extracted. It must be numeric with no decimal positions and can contain a field name, array element, table name, literal, or named constant. If the length is not specified, the rest of the base string (starting from the start location) is returned.

Factor 2 must contain either the base string, or the base string followed by a colon, followed by the start position. The base string must contain a field name, array element, named constant, data structure name, table name, or literal. The start position must be numeric with zero decimal positions, and can contain a field name, array element, table name, literal or named constant. If the start position is not specified, SUBST starts in position 1 of the base string. For graphic or UCS-2 strings, the start position is measured in double bytes.

Note:

- The start position and the length of the substring to be extracted must be positive integers
- The start position must not be greater than the length
- The length must not be greater than the length of the base string from the start location.

The result field must be character, graphic, or UCS-2 and can contain a field name, array element, data structure, or table name. The result is left-justified. The result field's length should be at least as large as the length specified in factor 1. If the substring is longer than the field specified in the result field, the substring is truncated from the right. If the result field is variable-length, its length does not change. If you specify P as the operation extender, the result field is padded from the right with blanks after the operation.

Note:

- You cannot use figurative constants in factor 1, factor 2, or the result field
- Overlapping is allowed for factor 1 and the result field
- Overlapping is allowed for factor 2 and the result field.

To handle SUBST exceptions program status code 100), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "String Operations" on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The SUBST operation extracts the substring from factor 2 starting
C* at position 3 for a length of 2.  The value 'CD' is placed in the
C* result field TARGET.  Indicator 90 is not set on because no error
C* occurred.
C
C                   Z-ADD     3             T                2 0
C                   MOVEL     'ABCDEF'      String          10
C     2             SUBST     String:T      Target               90
C*
C* In this SUBST operation, the length is greater than the length
C* of the string minus the start position plus 1.  As a result,
C* indicator 90 is set on and the result field is not changed.
C
C                   MOVE      'ABCDEF'      String           6
C                   Z-ADD     4             T                1 0
C     5             SUBST     String:T      Result               90
C
C* In this SUBST operation, 3 characters are substringed starting
C* at the fifth position of the base string.  Because P is not
C* specified, only the first 3 characters of TARGET are
C* changed.  TARGET contains '123XXXXX'.
C
C                   Z-ADD     3             Length           2 0
C                   Z-ADD     5             T                2 0
C                   MOVE      'TEST123'     String           8
C                   MOVE      *ALL'X'       Target
C     Length        SUBST     String:T      Target           8
```

Figure 329. SUBST Operation

```
C*
C* This example is the same as the previous one except P
C* specified, and the result is padded with blanks.
C* TARGET equals '123bbbbb'.
C
C                   Z-ADD     3             Length            2 0
C                   Z-ADD5                  T                 2 0
C                   MOVE      'TEST123'     String            8
C                   MOVE      *ALL'X'       Target
C     Length        SUBST(P)  String:T      Target            8
C
C
C*
C* In the following example, CITY contains the string
C* 'Toronto, Ontario'.  The SCAN operation is used to locate the
C* separating blank, position 9 in this illustration.  SUBST
C* without factor 1 places the string starting at position 10 and
C* continuing for the length of the string in field TCNTRE.
C* TCNTRE contains 'Ontario'.
C     ' '           SCAN      City          C
C                   ADD       1             C
C                   SUBST     City:C        TCntre
C*
C* Before the operations STRING='bbbJohnbbbbbb'.
C* RESULT is a 10 character field which contains 'ABCDEFGHIJ'.
C* The CHECK operation locates the first nonblank character
C* and sets on indicator 10 if such a character exists.  If *IN10
C* is on, the SUBST operation substrings STRING starting from the
C* first non-blank to the end of STRING.  Padding is used to ensure
C* that nothing is left from the previous contents of the result
C* field.  If STRING contains the value '   HELLO ' then RESULT
C* will contain the value 'HELLO     ' after the SUBST(P) operation.
C* After the operations RESULT='Johnbbbbbb'.
C
C     ' '           CHECK     STRING        ST                      10
C   10              SUBST(P)  STRING:ST     RESULT
```

Figure 330. SUBST Operation with the operation extender P

TAG (Tag)

Free-Form Syntax	(not allowed - use other operation codes, such as LEAVE, ITER, and RETURN)

Code	Factor 1	Factor 2	Result Field	Indicators		
TAG	Label					

The declarative TAG operation names the label that identifies the destination of a GOTO or CABXX operation. It can be specified anywhere within calculations.

A GOTO within a subroutine in the main procedure can be issued to a TAG within the same subroutine. A GOTO within a subroutine in a subprocedure can be issued to a TAG within the same subroutine, or within the body of the subprocedure.

Factor 1 must contain the name of the destination of a GOTO or CABxx operation. This name must be a unique symbolic name, which is specified in factor 2 of a GOTO operation or in the result field of a CABxx operation. The name can be used as a common point for multiple GOTO or CABxx operations.

Conditioning indicator entries (positions 9 through 11) are not allowed.

Branching to the TAG from a different part of the VRPG application may result in an endless loop.

See Figure 256 on page 585 for examples of the TAG operation.

For more information, see “Branching Operations” on page 352 or “Declarative Operations” on page 362.

TEST (Test Date/Time/Timestamp)

Free-Form Syntax	TEST{(EDTZ)} {*dtz-format*} *field-name*

Code	Factor 1 (*dtz-format*)	Factor 2	Result Field (*field-name*)	Indicators		
TEST (E)			Date/Time or Timestamp Field	_	ER	_
TEST (D E)	Date Format		Character or Numeric field	_	ER	_
TEST (E T)	Time Format		Character or Numeric field	_	ER	_
TEST (E Z)	Timestamp Format		Character or Numeric field	_	ER	_

The TEST operation allows you to test the validity of date, time, or timestamp fields prior to using them.

The *dtz-format* operand cannot be specified if the *field-name* operand is a date, time, or timestamp field. For information on the formats that you can use, see "Date Data" on page 119, "Time Data" on page 135, and "Timestamp Data" on page 137.

If *field-name* is a field declared as Date, Time, or Timestamp, *dtz-format* operand cannot be specified and the operation code extenders 'D', 'T', and 'Z' are not allowed.

If *field-name* is a field declared as character or numeric, then one of the operation code extenders 'D', 'T', or 'Z' must be specified.

Note: If *field-name* is a character field with no separators, the *dtz-format* operand must specify the date, time, or timestamp format followed by a zero.

- If the operation code extender includes 'D' (test Date):
  - *dtz-format* is optional and may be any of the Date formats. See "Date Data" on page 119.
  - If *dtz-format* is not specified, the format specified on the control specification with the DATFMT keyword is assumed. If this keyword is not specified, *ISO is assumed.
- If the operation code extender includes 'T' (test Time):
  - *dtz-format* is optional and may contain any of the valid Time formats. See "Time Data" on page 135.
  - If *dtz-format* is not specified, the format specified on the control specification with the TIMFMT keyword is the default. If this keyword is not specified, *ISO is assumed.

    Note: The *USA date format is not allowed with the operation code extender (T). The *USA date format has an AM/PM restriction that cannot be converted to numeric when a numeric result field is used.
- If the operation code extender includes 'Z' (test Timestamp), *dtz-format* is optional and may contain *ISO or *ISO0. See "Timestamp Data" on page 137.

Numeric fields are tested for valid digit portion of a Date, Time or Timestamp value. Character fields are tested for both valid digits and separators.

If the character or numeric field specified as the *field-name* operand is longer than required by the format being tested, extra data is ignored. For character data, only the leftmost data is used; for numeric data, only the rightmost data is used. For example, if the *dtz-format* operand is *MDY for a test of a numeric date, only the rightmost 6 digits of the *field-name* operand are examined.

For the test operation, either the operation code extender 'E' or an error indicator ER must be specified, but not both. If the content of the *field-name* operand is not valid, program status code 112 is signaled. Then, the error indicator is set on or the %ERROR built-in function is set to return '1' depending on the error handling method specified. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see "Date Operations" on page 359 or "Test Operations" on page 378.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D*
D Datefield       S               D   DATFMT(*JIS)
D Num_Date        S              6P 0 INZ(910921)
D Char_Time       S              8    INZ('13:05 PM')
D Char_Date       S              6    INZ('041596')
D Char_Tstmp      S             20    INZ('19960723140856834000')
D Char_Date2      S              9A   INZ('402/10/66')
D Char_Date3      S              8A   INZ('2120/115')
D*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  Indicator 18 will not be set on since the character field is a
C*  valid *ISO timestamp field, without separators.
C*
C     *ISO0         TEST (Z)               Char_Tstmp             18
C*
C*  Indicator 19 will not be set on since the character field is a
C*  valid *MDY date, without separators.
C*
C     *MDY0         TEST (D)               Char_Date              19
C*
C*  %ERROR will return '1', since Num_Date is not *DMY.
C*
C     *DMY          TEST (DE)              Num_Date
C*
C*  No Factor 1 since result is a D data type field
C*  %ERROR will return '0', since the field
C*  contains a valid date
C*
C                   TEST (E)               Datefield
C*
C* In the following test,  %ERROR will return '1' since the
C* Timefield does not contain a valid USA time
C*
C     *USA          TEST (ET)              Char_Time
C*
C*  In the following test, indicator 20 will be set on since the
C*  character field is a valid *CMDY, but there are separators.
C*
C     *CMDY0        TEST (D)               char_date2             20
C*
C*  In the following test, %ERROR will return '0' since
C*  the character field is a valid *LONGJUL date.
C*
C     *LONGJUL      TEST (DE)              char_date3
```

Figure 331. TEST (D/T/Z) Example

TESTB (Test Bit)

Free-Form Syntax	(not allowed - use the %BITAND built-in function. See Figure 138 on page 413.)

Code	Factor 1	Factor 2	Result Field	Indicators		
TESTB		Bit numbers	Character field	OF	ON	EQ

The TESTB operation compares the bits identified in factor 2 with the corresponding bits in the field named as the result field. The result field must be a one-position character field. Resulting indicators in positions 71 through 76 reflect the status of the result field bits. Factor 2 is always a source of bits for the result field.

Factor 2 can contain:

- Bit numbers 0-7: From 1 to 8 bits can be tested per operation. The bits to be tested are identified by the numbers 0 through 7. (0 is the leftmost bit.) The bit numbers must be enclosed in apostrophes. For example, to test bits 0, 2, and 5, enter '025' in factor 2.
- Field name: You can specify the name of a one-position character field, table name, or array element in factor 2. The bits that are on in the field, table name, or array element are compared with the corresponding bits in the result field; bits that are off are not considered. The field specified in the result field can be an array element if each element of the array is a one-position character field.
- Hexadecimal literal or named constant: You can specify a 1-byte hexadecimal literal or hexadecimal named constant. Bits that are on in factor 2 are compared with the corresponding bits in the result field; bits that are off are not considered.

Figure 332 on page 704 illustrates uses of the TESTB operation.

Indicators assigned in positions 71 through 76 reflect the status of the result field bits. At least one indicator must be assigned, and as many as three can be assigned for one operation. For TESTB operations, the resulting indicators are set on as follows:

- Positions 71 and 72: An indicator in these positions is set on if the bit numbers specified in factor 2 or each bit that is on in the factor 2 field is off in the result field. All of the specified bits are equal to off.
- Positions 73 and 74: An indicator in these positions is set on if the bit numbers specified in factor 2 or the bits that are on in the factor 2 field are of mixed status (some on, some off) in the result field. At least one the specified bits is on.

  Note: If only one bit is to be tested, these positions must be blank. If a field name is specified in factor 2 and it has only one bit on, an indicator in positions 73 and 74 is not set on.
- Positions 75 and 76: An indicator in these positions is set on if the bit numbers specified in the factor 2 or each bit that is on in factor 2 field is on in the result field. All of the specified bits are equal to on.

  Note: If the field in factor 2 has no bits on, then no indicators are set on.

For more information, see "Bit Operations" on page 352 or "Test Operations" on page 378.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The field bit settings are FieldF = 00000001, and FieldG = 11110001.
C*
C*  Indicator 16 is set on because bit 3 is off (0) in FieldF.
C*  Indicator 17 is set off.
C                   TESTB     '3'           FieldF               16  17
C*
C*  Indicator 16 is set on because both bits 3 and 6 are off (0) in
C*  FieldF.  Indicators 17 and 18 are set off.
C                   TESTB     '36'          FieldF               161718
C*
C*  Indicator 17 is set on because bit 3 is off (0) and bit 7 is on
C*  (1) in FieldF.  Indicators 16 and 18 are set off.
C                   TESTB     '37'          FieldF               161718
C*
C*  Indicator 17 is set on because bit 7 is on (1) in FieldF.
C*  Indicator 16 is set off.
C                   TESTB     '7'            FieldF              16  17
C*
C*  Indicator 17 is set on because bits 0,1,2, and 3 are off (0) and
C*  bit 7 is on (1).  Indicators 16 and 18 are set off.
C                   TESTB     FieldG         FieldF              161718
C*
C*  The hexadecimal literal X'88' (10001000) is used in factor 2.
C*  Indicator 17 is set on because at least one bit (bit 0) is on
C*  Indicators 16 and 18 are set off.
C                   TESTB     X'88'          FieldG              161718
```

Figure 332. TESTB Operation

TESTN (Test Numeric)

Free-Form Syntax	(not allowed - rather than testing the variable before using it, code the usage of the variable in a MONITOR group and handle any errors with ON-ERROR. See Error-Handling Operations.)

Code	Factor 1	Factor 2	Result Field	Indicators		
TESTN			Character field	NU	BN	BL

The TESTN operation tests a character result field for the presence of zoned decimal digits and blanks.

The result field must be a character field. To be considered numeric, each character in the field, except the right-most character, must contain a hexadecimal 3 zone and a digit (0 through 9). The right-most character is numeric if it contains a hexadecimal 0 through 9, or an A to F zone, and a digit (0 through 9). As a result of the test, resulting indicators are set on as follows:

- Positions 71 and 72: The result field contains numeric characters.
- Positions 73 and 74: The result field contains both numeric characters and at least one leading blank. For example, the values b123 or bb123 set this indicator on. However, the value b1b23 is not a valid numeric field because of the embedded blanks, so this value does not set this indicator on.

  Note: An indicator cannot be specified in positions 73 and 74 when a field of length one is tested because the character field must contain at least one numeric character and one leading blank.
- Positions 75 and 76: The result field contains all blanks.

The same indicator can be used for more than one condition. If any of the conditions exist, the indicator is set on.

The TESTN operation may be used to validate fields before they are used in operations where their use may cause undesirable results or exceptions (for example, arithmetic operations).

For more information, see “Test Operations” on page 378.

TESTZ (Test Zone)

Free-Form Syntax	(not allowed - use the %BITAND builtin function with X'F0' to isolate the zone part of the character)

Code	Factor 1	Factor 2	Result Field	Indicators		
TESTZ			Character field	+	–	

The TESTZ operation tests the zone of the leftmost character in the result field. The result field must be a character field.

Resulting indicators are set on according to the results of the test. You must specify at least one resulting indicator positions 71 through 74. Any character with a positive zone (hexadecimal 0, 1, 2, 3, 8, 9, A, B) sets on the indicator in positions 71 and 72. Any character with a minus zone (hexadecimal 4, 5, 6, 7, C, D, E, F) sets on the indicator in positions 73 and 74.

Note: The positive/negative zone depends on the value of the second bit. The values 3 and 7 are the preferred values for the sign, however, the other values that are listed can be used.

For more information, see "Test Operations" on page 378.

TIME (Time of Day)

Free-Form Syntax	(not allowed – use the %DATE, %TIME, and %TIMESTAMP built-in functions)

Code	Factor 1	Factor 2	Result Field	Indicators		
TIME	Alias name		Target field			

The TIME operation accesses the system time and, if specified, the system date at any time during program processing. The system time and date can be retrieved from either an iSeries server or from the workstation. The system time is based on the 24-hour clock.

If factor 1 is specified, it must contain a literal which is the alias name of a server. If factor 1 is not specified, the time from the workstation is retrieved.

The result field must specify the name of a 6-, 12-, or 14-digit numeric field (no decimal positions). The time of day or the time of day and the system date are written into the result field.

Result Field	Value Returned	Format
6-digit Numeric	Time	hhmmss
12-digit Numeric	Time and Date	hhmmssDDDDDD
14-digit Numeric	Time and Date	hhmmssDDDDDDDD
Time	Time	Format of Result
Date	Date	Format of Result
Timestamp	Timestamp	*ISO
Note: If the result field is a numeric field and the system date is included, it is placed in positions 7 through 12 of the result field. The date format is *YMD.		

To access the time of day only, specify the result field as a 6-digit numeric field. To access both the time of day and the system date, specify the result field as a 12- (2-digit year portion) or 14-digit (4-digit year portion) numeric field. The time of day is always placed in the first six positions of the result field in the following format:

hhmmss (hh=hours, mm=minutes, and ss=seconds)

The date format depends on the date format job attribute DATFMT and can be mmddyy, ddmmyy, yymmdd, or Julian. The Julian format for 2-digit year portion contains the year in positions 7 and 8, the day (1 through 366, right-adjusted, with zeros in the unused high-order positions) in positions 9 through 11, and 0 in position 12. For 4-digit year portion, it contains the year in positions 7 through 10, the day (1 through 366, right-adjusted, with zeros in the unused high-order positions) in positions 11 through 13, and 0 in position 14.

If the Result field is a Timestamp field, the last 3 digits in the microseconds part is always 000.

The special fields UDATE and *DATE contain the job date. These values are not updated when midnight is passed, or when the job date is changed during the running of the program.

TIME (Time of Day)

For more information, see “Information Operations” on page 366.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  When the TIME operation is processed (with a 6-digit numeric
C*  field), the current time (in the form hhmmss) is placed in the
C*  result field CLOCK.  The TIME operation is based on the 24-hour
C*  clock, for example, 132710.  (In the 12-hour time system, 132710
C*  is 1:27:10 p.m.)  CLOCK can then be specified in the output
C*  specifications.
C                   TIME                    Clock             6 0
C*  When the TIME operation is processed (with a 12-digit numeric
C*  field), the current time and day is placed in the result field
C*  TIMSTP.  The first 6 digits are the time, and the last 6 digits
C*  are the date; for example, 093315121579 is 9:33:15 a.m. on
C*  December 15, 1979.  TIMSTP can then be specified in the output
C*  specifications.
C                   TIME                    TimStp           12 0
C                   MOVEL     TimStp        Time              6 0
C                   MOVE      TimStp        SysDat            6 0
C*  This example duplicates the 12-digit example above but uses a
C*  14-digit field.  The first 6 digits are the time, and the last
C*  8 digits are the date;   for example, 13120001101992
C*  is 1:12:00 p.m. on January 10, 1992.
C*  TIMSTP can then be specified in the output specifications.
C                   TIME                    TimStp           14 0
C                   MOVEL     TimStp        Time              6 0
C                   MOVE      TimStp        SysDat            8 0
```

Figure 333. TIME Operation

UNLOCK (Unlock a Data Area or Release a Record)

Free-Form Syntax	UNLOCK{(E)} *name*

Code	Factor 1	Factor 2	Result Field	Indicators		
UNLOCK (E)		name (file or data area)		_	ER	_

The UNLOCK operation unlocks data areas and releases record locks.

To handle UNLOCK exceptions (program status codes 401-421, 431, and 432), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

Positions 71,72,75 and 76 must be blank.

For further rules for the UNLOCK operation, see "Data-Area Operations" on page 358.

Unlocking data areas

The *name* operand must be the name of the data area to be unlocked or the reserved word *DTAARA.

When *DTAARA is specified, all data areas in the program that are locked are unlocked.

The data area must already be specified in the result field of an *DTAARA DEFINE statement or with the DTAARA keyword on the definition specification. If the UNLOCK operation is specified to an already unlocked data area, an error does not occur.

For more information, see "File Operations" on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  TOTAMT, TOTGRS, and TOTNET are defined as data areas in the
C*  system.  The IN operation retrieves all the data areas defined in
C*  the program.  The program processes calculations, and
C*  then unlocks the data areas. The data areas can then be used
C*  by other programs.
C*
C     *LOCK         IN        *DTAARA
C                   :
C                   :
C                   UNLOCK    *DTAARA
C     *DTAARA       DEFINE                TOTAMT             8 2
C     *DTAARA       DEFINE                TOTGRS            10 2
C     *DTAARA       DEFINE                TOTNET            10 2
```

Figure 334. Data area unlock operation

Releasing record locks

The UNLOCK operation unlocks the most recently locked record in an update disk file.

The *name* operand must be the name of the UPDATE disk file.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FFilename++IT.A.FRlen+......A.Device+.Keywords++++++++++++++++++++++++++++++
F*
FUPDATA    UF   E             DISK    REMOTE
F*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  Assume that the file UPDATA contains record format VENDOR.
C*  A record is read from UPDATA. Since the file is an update
C*  file, the record is locked. *IN50 is set somewhere else in
C*  the program to control whether an UPDATE should take place.
C*  Otherwise the record is unlocked using the UNLOCK operation.
C*  Note that factor 2 of the UNLOCK operation is the file name,
C*  UPDATA, not the record format, VENDOR.
C*
C                   READ      VENDOR                                 12
C                   :
C     *IN50         IFEQ      *ON
C                   UPDATE    VENDOR
C                   ELSE
C                   UNLOCK    UPDATA                               99
C                   ENDIF
```

Figure 335. Record unlock operation

UPDATE (Modify Existing Record)

Free-Form Syntax	UPDATE{(E)} *name* {*data-structure*	%FIELDS(*name*{:*name...*})}

Code	Factor 1	Factor 2	Result Field	Indicators		
UPDATE (E)		name (file, record format, or subfile)	data-structure	_	ER	_

The UPDATE operation modifies the last locked record retrieved for processing from an update disk file or subfile. No other operation should be performed on the file between the input operation that retrieved and locked the record, and the UPDATE operation.

Operations such as READ, READC, READE, READP, READPE, and CHAIN retrieve and lock a record. If these input operations are not successful, the record is not locked and UPDATE cannot be issued. The record must be read again with the default of a blank operation extender to specify a lock request.

After a successful UPDATE operation, the next sequential input operation retrieves the record following the updated record.

Consecutive UPDATE operations to the same file or record are not valid. Intervening successful read operations must be issued to position to and lock the record to be updated.

The *name* operand must be the name of a file, subfile, or record format to be updated. If a file name is specified, the file must be program described. If a record format name is specified, the file must be externally described. The record format name must be the name of the last record read from the file; otherwise an error occurs.

If the data-structure operand is specified, the record is updated directly from the data structure. The data structure must conform to the rules below:

1. If the *data-structure* operand is specified, the record is updated directly from the data structure.
2. If *name* refers to a program-described file (identified by an F in Position 22 of the file description specification), the data structure can be any data structure of the same length as the file's declared record length.
3. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT).
4. See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the data structure and the file.

A list of the fields to update can be specified using %FIELDS. The parameter to %FIELDS is a list of the field names to update. See the example at the end of "%FIELDS (Fields to update)" on page 442 for an illustration of updating fields.

To handle UPDATE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "File Exception/Errors" on page 41.

Note: If some but not all fields in a record are to be updated, either use the output specifications without an UPDATE operation or use %FIELDS to identify which fields to update.

See “Database Null Value Support” on page 137 for information on reading records with null-capable fields.

For more information, see “File Operations” on page 363.

WHEN (When True Then Select)

Free-Form Syntax	WHEN{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2				
WHEN (M/R)		indicator-expression				

TheWHEN operation code is similar to the WHENxx operation code in that it controls the processing of lines in a SELECT operation. It differs in that the condition is specified by a logical expression in the *indicator-expression* operand. The operations controlled by the WHEN operation are performed when the *indicator-expression* is true. See Chapter 24, “Expressions,” on page 381 for details on expressions. For information on how operation extenders M and R are used, see “Precision Rules for Numeric Operations” on page 390

For more information, see “Compare Operations” on page 357 or “Structured Programming Operations” on page 376.

```
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++..
C*
C                   SELECT
C                   WHEN    *INKA
C                   :
C                   :
C                   :
C                   WHEN    NOT(*IN01) AND (DAY = 'FRIDAY')
C                   :
C                   :
C                   :
C                   WHEN    %SUBST(A:(X+4):3) = 'ABC'
C                   :
C                   :
C                   :
C                   OTHER
C                   :
C                   :
C                   :
C                   ENDSL
```

Figure 336. WHEN Operation

WHENxx (When True Then Select)

Free-Form Syntax	(not allowed - use the WHENoperation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
WHENxx	Comparand	Comparand				

The WHENxx operations of a select group determine where control passes after the SELECT operation is processed.

The WHENxx conditional operation is true if factor 1 and factor 2 have the relationship specified by xx. If the condition is true, the operations following the WHENxx are processed until the next WHENxx, OTHER, ENDSL, or END operation.

When performing the WHENxx operation remember:
- After the operation group is processed, control passes to the statement following the ENDSL operation.
- You can code complex WHENxx conditions using ANDxx and ORxx. Calculations are processed when the condition specified by the combined WHENxx, ANDxx, and ORxx operations is true.
- The WHENxx group can be empty.

Refer to “Compare Operations” on page 357 for xx values.

For more information, see “Structured Programming Operations” on page 376.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following example shows nested SELECT groups.  The employee
C* type can be one of 'C' for casual, 'T' for retired, 'R' for
C* regular, and 'S' for student.  Depending on the employee type
C* (EmpTyp), the number of days off per year (Days) will vary.
C*
C                   SELECT
C     EmpTyp        WHENEQ    'C'
C     EmpTyp        OREQ      'T'
C                   Z-ADD     0             Days
C     EmpTyp        WHENEQ    'R'
C*
C* When the employee type is 'R', the days off depend also on the
C* number of years of employment.  The base number of days is 14.
C* For less than 2 years, no extra days are added.  Between 2 and
C* 5 years, 5 extra days are added.  Between 6 and 10 years, 10
C* extra days are added, and over 10 years, 20 extra days are added.
C*
C                   Z-ADD     14            Days
C*
C* Nested select group.
C                   SELECT
C     Years         WHENLT    2
C     Years         WHENLE    5
C                   ADD       5             Days
C     Years         WHENLE    10
C                   ADD       10            Days
C                   OTHER
C                   ADD       20            Days
C                   ENDSL
C* End of nested select group.
C*
C     EmpTyp        WHENEQ    'S'
C                   Z-ADD     5             Days
C                   ENDSL
```

Figure 337. WHENxx Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C* Example of a SELECT group with complex WHENxx expressions.  Assume
C* that a record and an action code have been entered by a user.
C* Select one of the following:
C*   * When F3 has been pressed, do subroutine QUIT.
C*   * When action code(Acode) A (add) was entered and the record
C*     does not exist (*IN50=1), write the record.
C*   * When action code A is entered, the record exists, and the
C*     active record code for the record is D (deleted); update
C*     the record with active rec code=A.  When action code D is
C*     entered, the record exists, and the action code in the
C*     record (AcRec) code is A; mark the record as deleted.
C*   * When action code is C (change), the record exists, and the
C*     action code in the record (AcRec) code is A; update the record.
C*   * Otherwise, do error processing.
C*
C     RSCDE         CHAIN     FILE                              50
C                   SELECT
C     *INKC         WHENEQ    *ON
C                   EXSR      QUIT
C     Acode         WHENEQ    'A'
C     *IN50         ANDEQ     *ON
C                   WRITE     REC
C     Acode         WHENEQ    'A'
C     *IN50         ANDEQ     *OFF
C     AcRec         ANDEQ     'D'
C     Acode         OREQ      'D'
C     *IN50         ANDEQ     *OFF
C     AcRec         ANDEQ     'A'
C                   MOVE      Acode         AcRec
C                   UPDATE    REC
C     Acode         WHENEQ    'C'
C     *IN50         ANDEQ     *OFF
C     AcRec         ANDEQ     'A'
C                   UPDATE    REC
C                   OTHER
C                   EXSR      ERROR
C                   ENDSL
```

Figure 338. WHENxx Operation

WRITE (Create New Records)

Free-Form Syntax	WRITE{(E)} *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
WRITE (E)		name (file, record format, subfile, or window)	data-structure	_	ER	EOF

The WRITE operation writes data to a file or window.

Writing to a File

The WRITE operation writes a new record to a file.

The name operand must be the name of a full-procedural file or record format. A record format name is allowed only with an externally described file. A file name is required with a program described file.

If the data-structure operand is specified, the record is written directly from the data structure to the file. If name refers to a program described file (identified by an F in position 22 of the file description specification), the data structure is required and can be any data structure of the same length as the file's declared record length. If *name* refers to an externally-described file or a record format from an externally described file, the data structure must be a data structure defined with EXTNAME(...:*OUTPUT) or LIKEREC(...:*OUTPUT). See "File Operations" on page 363 for information on how to define the data structure and how data is transferred between the file and the data structure.

When records that use relative record numbers are written to a file, the field name specified on the RECNO File specification keyword (relative record number) must be updated so it contains the relative record number of the record to be written.

For local files, records are written to the end of the file. The RECNO keyword is ignored.

To add records to a remote DISK file using the WRITE operation, an A must be specified in position 20 of the file description specifications. For local files, records are added to the end of the file. See "Position 20 (File Addition)" on page 240 for more information.

To handle WRITE exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. An error occurs if overflow is reached to an externally described print file and no overflow indicator has been specified on the File description specification. For more information on error handling, see "File Exception/Errors" on page 41.

You can specify an indicator in positions 75-76 to signal whether an end of file occurred (subfile is filled) on the WRITE operation. The indicator is set on (an EOF condition), or off, every time the WRITE operation is performed. This information can also be obtained from the %EOF built-in function, which returns '1' if an EOF condition occurs and '0' otherwise.

See "Database Null Value Support" on page 137 for information on adding records with null-capable fields containing null values.

For more information, see "File Operations" on page 363.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C*  The WRITE operation writes the fields in the data structure
C*  DS1 to the file, FILE1.
C*
C                   WRITE     FILE1         DS1
```

Figure 339. WRITE Operation for Files

Writing to a Window

If the *name* operand is a window name, the WRITE operation sets the attributes of static text and field parts on the window. The attribute for entry parts is TEXT. The attribute for static text parts is LABEL.

The *data-structure* operand and the operation extender cannot be specified.

When a window is written, the values stored in corresponding fields are used are used to set the attributes of the entry field parts and the static text parts. After the WRITE operation, the values stored in the fields match the values on the display. If there are many static text and entry fields, use the WRITE operation rather than multiple SETATRs. For example, if window INVENTORY contains the entry field parts ENT0000B and ENT0000C a WRITE of the window performs the equivalent to the following:

```
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments++++++
CSRN01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++++Comments++++++
C                   EVAL      %setatr('inventory':'ent0000B':'text') = ENT0000B
C                   EVAL      %setatr('inventory':'ent0000C':'text') = ENTd000C
```

Figure 340. WRITE Operation for Windows

Writing to a Subfile

If the *name* operand is a subfile part name, the WRITE operation adds a new record to the subfile. Records written to a subfile part are added to the end of the subfile.

The *data-structure* operand cannot be specified.

XFOOT (Summing the Elements of an Array)

Free-Form Syntax	(not allowed - use the %XFOOT built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
XFOOT (H)		Array name	Sum	+	–	Z

XFOOT adds the elements of an array together and places the sum into the field specified as the result field. Factor 2 contains the name of the array.

If half-adjust is specified, the rounding occurs after all elements are summed and before the results are moved into the result field. If the result field is an element of the array specified in factor 2, the value of the element before the XFOOT operation is used to calculate the total of the array.

If the array is float, XFOOT will be performed as follows: when the array is in descending sequence, the elements will be added together in reverse order. Otherwise, the elements will be added together starting with the first elements of the array.

For further rules for the XFOOT operation, see “Arithmetic Operations” on page 348 or “Array Operations” on page 351.

Figure 120 on page 351 contains an example of the XFOOT operation.

XLATE (Translate)

Free-Form Syntax	(not allowed - use the %XLATE built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
XLATE (E P)	From:To	Source String:start	Target String	_	ER	_

The XLATE operation translates characters in the source string (factor 2) that match the From string to the corresponding characters in the To string. XLATE starts translating the source at the position specified in factor 2 and continues character by character, from left to right. If a character in the source string exists in the From string, the corresponding character in the To string is placed in the result field. Any characters in the source field before the starting position are placed unchanged in the result field.

Note:

- The From, To, Source, and Target strings must all be of the same type, either all character, all graphic, or all UCS-2. As well, their CCSIDs must be the same, unless, in the case of graphic fields, CCSID(*GRAPH : *IGNORE) was specified on the Control Specification.
- Figurative constants cannot be used in factor 1, factor 2, or result fields. No overlapping in a data structure is allowed for factor 1 and the result field, or factor 2 and the result field.

Factor 1 must contain the From string, followed by a colon, followed by the To string. The From and To strings can contain a field name, array element, named constant, data structure name, literal, or table name. If a character in the From string is duplicated, the first occurrence (leftmost) is used.

Factor 2 must contain either the source string or the source string followed by a colon and the start position. The source string portion of factor 2 can contain a field name, array element, named constant, data structure name, data structure subfield, literal, or table name. If the operation uses graphic or UCS-2 data, the start position refers to double-byte characters. The start position position of factor 2 must be numeric with no decimal positions and can be a named constant, array element, field name, literal, or table name. If no start position is specified, the default is 1.

The result field can be a field, an array element, a data structure, or a table. The length of the result field should be as large as the source string specified in factor 2. If the result field is larger than the source string, the result is left adjusted. If the result field is larger than the source string and the operation extender P is specified, the result is padded on the right with blanks after the translation. If the result field is shorter than the source string, the result field contains the leftmost part of the translated source.

Note: If the result field is graphic and the operation extender P is specified, then graphic blanks are be used. If the result field is UCS-2 and P is specified, UCS-2 blanks will be used.

To handle XLATE exceptions (program status code 100), either the operation code extender 'E' or an error indicator ER can be specified, but not both. For more information on error handling, see "Program Exception and Errors" on page 51.

For more information, see “String Operations” on page 375.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*
C* The following translates the blank in NUMBER to '-'.  The result
C* in RESULT will be '999-9999'.
C*
C                   MOVE      '999 9999'    Number            8
C     ' ':'-'       XLATE     Number        Result            8
```

Figure 341. XLATE Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D*
D* In the following example, all values in STRING are translated to
D* uppercase.  As a result, RESULT='RPG DEP'.
D*
D Up              C                   'ABCDEFGHIJKLMNOPQRS-
D                                     'TUVWXYZ'
D Lo              C                   'abcdefghijklmnopqrs-
D                                     'tuvwxyz'
C*
CSRN01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C
C                   MOVE      'rpg dep'     String            7
C     Lo:Up         XLATE     String        Result               90
C*
C* In the following example all values in the string are translated
C* to lowercase.  As a result, RESULT='rpg dep'.
C*
C                   MOVE      'RPG DEP'     String            7
C     Up:Lo         XLATE     String        Result               90
```

Figure 342. XLATE Operation With Named Constants

Z-ADD (Zero and Add)

Free-Form Syntax	(not allowed - use the EVAL operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
Z-ADD (H)		Addend	Sum	+	–	Z

Factor 2 is added to a field of zeros. The sum is placed in the result field. Factor 2 must be numeric and can contain an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric and can contain an array, array element, subfield, or table name.

Half-adjust can be specified.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the Z-ADD operation.

Z-SUB (Zero and Subtract)

Free-Form Syntax	(not allowed - use the EVAL operation code)

Code	Factor 1	Factor 2	Result Field	Indicators		
Z-SUB (H)		Subtrahend	Difference	+	–	Z

Factor 2 is subtracted from a field of zeros. The difference, which is the negative of factor 2, is placed in the result field.

Factor 2 must be numeric and can contain an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric and can contain an array, array element, subfield, or table name.

Half-adjust can be specified.

"Arithmetic Operations" on page 348 describes the general rules for specifying arithmetic operations.

Figure 120 on page 351 shows examples of the Z-SUB operation.

Part 5. Appendixes

Appendix A. Restrictions

Function	Restriction
Array/table input record length for compile time	Maximum length is 100
Character field length	Maximum length is 65535 bytes
Graphic or UCS-2 field length	Maximum length is 32766 bytes
Data structure (named) length	Maximum of 65535
Data structure (unnamed) length	Maximum of 9999999
Data structure occurrences (number of)	Maximum of 32767 per data structure
Edit Word	Maximum length of 115
Elements in an array/table (DIM keyword on the definition specifications)	Maximum of 32767 per array/table
Levels of nesting in structured groups	Maximum of 100
Named Constant or Literal	Maximum length of 1024 bytes for a character, hexadecimal, graphic, or UCS-2 literal and 31 digits with 31 decimal positions for a numeric literal.
Parameters passed to remote OS/400 programs (CALL)	Maximum of 25
Total size of all parameters passed to remote OS/400 programs (CALL)	Maximum of 32767 bytes
Parameters passed to called functions (CALLB)	Maximum of 399
Parameters passed to local EXEs, CMDs, COMS, and BATs (CALLP)	Maximum of 20 or a total of 1024 bytes.
Printer files	Maximum of 8 per program
Printing lines per page	Minimum of 2; maximum of 255
Program status data structure	Only 1 allowed per program
Record length	Maximum length is 99999.[1]
Note: [1]Any device record size restraints override this value.	

Appendix B. Collating Sequences

The following tables list both the EBCDIC and ASCII collating sequences.

EBCDIC Collating Sequence

Table 63. EBCDIC Collating Sequence

Ordinal Number	Symbol	Meaning	Decimal Representation	Hex Representation
65	␣	Space	64	40
. . .				
75	¢	Cent sign	74	4A
76	.	Period, decimal point	75	4B
77	<	Less than sign	76	4C
78	(	Left parenthesis	77	4D
79	+	Plus sign	78	4E
80	v	Vertical bar, Logical OR	79	4F
81	&	Ampersand	80	50
. . .				
91	!	Exclamation point	90	5A
92	$	Dollar sign	91	5B
93	*	Asterisk	92	5C
94	)	Right parenthesis	93	5D
95	;	Semicolon	94	5E
96	¬	Logical NOT	95	5F
97	–	Minus, hyphen	96	60
98	/	Slash	97	61
. . .				
107	¦	Split vertical bar	106	6A
108	,	Comma	107	6B
109	%	Percent sign	108	6C
110	_	Underscore	109	6D
111	>	Greater than sign	110	6E
112	?	Question mark	111	6F
. . .				
122	`	Accent grave	121	79
123	:	Colon	122	7A
124	#	Number sign, pound sign	123	7B
125	@	At sign	124	7C
126	’	Apostrophe, prime sign	125	7D

Table 63. EBCDIC Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Representation	Hex Representation
127	=	Equal sign	126	7E
128	"	Quotation marks	127	7F
. . .				
130	a		129	81
131	b		130	82
132	c		131	83
133	d		132	84
134	e		133	85
135	f		134	86
136	g		135	87
137	h		136	88
138	i		137	89
. . .				
146	j		145	91
147	k		146	92
148	l		147	93
149	m		148	94
150	n		149	95
151	o		150	96
152	p		151	97
153	q		152	98
154	r		153	99
. . .				
162	~	Tilde	161	A1
163	s		162	A2
164	t		163	A3
165	u		164	A4
166	v		165	A5
167	w		166	A6
168	x		167	A7
169	y		168	A8
170	z		169	A9
. . .				
193	{	Left brace	192	C0
194	A		193	C1
195	B		194	C2
196	C		195	C3
197	D		196	C4

Table 63. EBCDIC Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Representation	Hex Representation
198	E		197	C5
199	F		198	C6
200	G		199	C7
201	H		200	C8
202	I		201	C9
. . .				
209	}	Right brace	208	D0
210	J		209	D1
211	K		210	D2
212	L		211	D3
213	M		212	D4
214	N		213	D5
215	O		214	D6
216	P		215	D7
217	Q		216	D8
218	R		217	D9
. . .				
225	\	Left slash	224	E0
. . .				
227	S		226	E2
228	T		227	E3
229	U		228	E4
230	V		229	E5
231	W		230	E6
232	X		231	E7
233	Y		232	E8
234	Z		233	E9
. . .				
241	0		240	F0
242	1		241	F1
243	2		242	F2
244	3		243	F3
245	4		244	F4
246	5		245	F5
247	6		246	F6
248	7		247	F7
249	8		248	F8
250	9		249	F9

ASCII Collating Sequence

Table 64. ASCII Collating Sequence

Ordinal Number	Symbol	Meaning	Decimal Represen- tation	Hex Represen- tation
1		Null	0	0
. . .				
33	b	Space	32	20
34	!	Exclamation point	33	21
35	"	Quotation mark	34	22
36	#	Number sign	35	23
37	$	Dollar sign	36	24
38	%	Percent sign	37	25
39	&	Ampersand	38	26
40	'	Apostrophe, prime sign	39	27
41	(	Opening parenthesis	40	28
42	)	Closing parenthesis	41	29
43	*	Asterisk	42	2A
44	+	Plus sign	43	2B
45	,	Comma	44	2C
46	–	Hyphen, minus	45	2D
47	.	Period, decimal point	46	2E
48	/	Slant	47	2F
49	0		48	30
50	1		49	31
51	2		50	32
52	3		51	33
53	4		52	34
54	5		53	35
55	6		54	36
56	7		55	37
57	8		56	38
58	9		57	39
59	:	Colon	58	3A
60	;	Semicolon	59	3B
61	<	Less than sign	60	3C
62	=	Equal sign	61	3D
63	>	Greater than sign	62	3E

Table 64. ASCII Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Represen- tation	Hex Represen- tation
64	?	Question mark	63	3F
65	@	Commercial At sign	64	40
66	A		65	41
67	B		66	42
68	C		67	43
69	D		68	44
70	E		69	45
71	F		70	46
72	G		71	47
73	H		72	48
74	I		73	49
75	J		74	4A
76	K		75	4B
77	L		76	4C
78	M		77	4D
79	N		78	4E
80	O		79	4F
81	P		80	50
82	Q		81	51
83	R		82	52
84	S		83	53
85	T		84	54
86	U		85	55
87	V		86	56
88	W		87	57
89	X		88	58
90	Y		89	59
91	Z		90	5A
92	[	Opening bracket	91	5B
93	\	Reverse slant	92	5C
94	]	Closing bracket	93	5D
95	^	Caret	94	5E
96	_	Underscore	95	5F
97	`	Grave Accent	96	60
98	a		97	61
99	b		98	62
100	c		99	63
101	d		100	64

Table 64. ASCII Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Represen- tation	Hex Represen- tation
102	e		101	65
103	f		102	66
104	g		103	67
105	h		104	68
106	i		105	69
107	j		106	6A
108	k		107	6B
109	l		108	6C
110	m		109	6D
111	n		110	6E
112	o		111	6F
113	p		112	70
114	q		113	71
115	r		114	72
116	s		115	73
117	t		116	74
118	u		117	75
119	v		118	76
120	w		119	77
121	x		120	78
122	y		121	79
123	z		122	7A
124	{	Opening brace	123	7B
125	¦	Split vertical bar	124	7C
126	}	Closing brace	125	7D
127	~	Tilde	126	7E

Appendix C. Supported CCSID Values

The following list contains the supported CCSID values for conversions to and from UCS-2 values. Conversion between unicode CCSIDs is not supported.

Note: The CCSIDs 932, 936, and 949 are converted as follows:

```
CCSID   Maps to
932      943
936     1386
949     1363
```

037	395	870	939	1018	1141	4951
256	420	871	941	1019	1142	4952
259	423	874	942	1025	1143	4960
273	424	875	943	1026	1144	5037
274	437	880	946	1027	1145	5039
277	500	891	947	1028	1146	5048
278	813	895	948	1038	1147	5049
280	819	896	950	1040	1148	5067
282	829	897	951	1041	1149	5142
284	833	903	971	1042	1250	5346
285	834	904	952	1043	1251	5347
287	835	905	955	1046	1252	5348
290	836	907	960	1047	1253	5349
293	837	909	961	1050	1254	5350
297	838	910	963	1051	1255	5351
300	850	912	964	1088	1256	5352
301	851	913	933	1089	1257	5353
361	852	914	949	1092	1275	5354
363	855	915	970	1097	1276	5478
367	856	916	1004	1098	1277	8612
382	857	918	1006	1112	1350	9030
383	858	919	1008	1114	1351	9056
385	860	920	1009	1115	1363	9066
386	861	921	1010	1116	1364	9145
387	862	922	1011	1117	1380	28709
388	863	923	1012	1118	1381	33722
389	864	924	1013	1119	1382	
391	865	927	1014	1122	1383	
392	866	930	1015	1123	1386	
393	868	935	1016	1124	1388	
394	869	937	1017	1140	4948	

Appendix D. Comparing RPG Compilers

The VisualAge RPG language is based on the RPG IV language. It has been enhanced so that you can develop and run applications with a graphical user interface in a client/server environment.

There are cases where certain features are not supported for VisualAge RPG. For example, there is no RPG cycle for VisualAge RPG. Because the RPG cycle is not supported, any features associated with this, such as, control breaks and matching fields, are also not supported.

In order to take advantage of the workstation application development environment, features have been added to VisualAge RPG (for example, operation codes such as SHOWWIN are used to display an application's windows) or existing ILE RPG for AS/400 features have been modified (for example, with the /COPY compiler directive, you copy either OS/400 files or workstation files).

This appendix summarizes the differences between ILE RPG and VisualAge RPG.

RPG Cycle

Since the RPG cycle is not supported for VisualAge RPG, indicators associated with the cycle are not supported. Entries on the specifications associated with the RPG cycle are also not supported.

VisualAge RPG Indicators

The following indicators are supported for VisualAge RPG. For a list of indicators not supported by VisualAge RPG, see "Unsupported Indicators."

Record identifying indicators
01 - 99, LR

Field indicators
01 - 99

Resulting indicators
01 - 99, LR

Unsupported Indicators

The following usage for indicators is not supported:

Overflow indicators
*INOA - *INOG, *INOV, *IN01 - *IN99, 1P

Record identifying indicators
H1 - H9, L1 - L9, U1 - U8, RT

Control level indicators
L1 - L9

Field indicators
H1 - H9, U1 - U8, RT

Resulting indicators
H1 - H9, OA - OG, OV, L1 - L9, U1 - U8,KA - KN, KP - KY, RT

File conditioning
The EXTIND keyword is not supported on the VisualAge RPG file description specifications. This means that you cannot use indicators for file conditioning.

Unsupported Words

The following describes words with special functions and reserved words that are not supported for VisualAge RPG.

- *ALTSEQ, *EQUATE, *FILE,
- User date: VisualAge RPG programs cannot be run in batch. Therefore, any of the rules for user date and batch programs do not apply to VisualAge RPG programs.

For a description of VisualAge RPG words, see Chapter 1, "Symbolic Names and Reserved Words," on page 3.

Compiler Directives

The /COPY compiler directive includes records from another file. This file can exist on your workstation or on an iSeries server. The records are inserted where the /COPY statement occurs and are compiled with the program For more information, see "/COPY or /INCLUDE)" on page 11.

Error and Exception Handling

Error and exception handling for VisualAge RPG applications includes support for handling components and the application's graphical user interface. For more information, see Chapter 4, "Working with Components," on page 31 and "Event Error Handling" on page 59.

Data

Data Types and Data Formats

The following summarizes the differences between the ILE RPG for AS/400 and VisualAge RPG data types and formats. For a description of the data types and formats supported for VisualAge RPG see Chapter 9, "Data Types and Data Formats," on page 103.

Binary format
Binary data is reordered when data is used between the server and the client.

Basing pointer data type
The length of an ILE RPG for AS/400 basing pointer field is 16 bytes long and must be aligned on a 16 byte boundary. The length of a VisualAge RPG basing pointer field is 4 bytes long and must be aligned on a 4 byte boundary.

Packed decimal format
ILE RPG for AS/400 uses hexadecimal F for positive numbers and hexadecimal D for negative numbers. VisualAge RPG uses hexadecimal C for positive numbers and hexadecimal D for negative numbers. VisualAge RPG also supports hexadecimal A, E, and F for positive numbers and hexadecimal B for negative numbers.

Procedure pointer data type

The length of an ILE RPG for AS/400 procedure pointer field is 16 bytes long and must be aligned on a 16 byte boundary. The length of a VisualAge RPG procedure pointer field is 4 bytes long and must be aligned on a 4 byte boundary.

Zoned-Decimal Format

ILE RPG for AS/400 uses hexadecimal F for positive numbers and hexadecimal D for negative numbers. VisualAge RPG uses hexadecimal 3 for positive numbers and hexadecimal 7 for negative numbers. VisualAge RPG also supports hexadecimal 0, 1, 2, 8, 9, A, and B for positive numbers and hexadecimal 4, 5, 6, C, D, E, and F for negative numbers.

Literals and Named Constants

The following describes the differences between the ILE RPG for AS/400 and VisualAge RPG literals and named constants. For a description of the data types and formats supported for VisualAge RPG, see Chapter 10, “Literals and Named Constants,” on page 149.

Graphic literals

An ILE RPG for AS/400 graphic character has the form G'oK1K2i' where **o** and **i** are the shift-out and shift-in characters. The shift-out and shift-in characters are not used with VisualAge RPG graphic characters. The form is G'K1K2'.

Figurative constants

ILE RPG for AS/400 figurative constants can use the shift-out and shift-in characters, for example, ALLG'oK1K2i'. The shift-out and shift-in characters are not used for VisualAge RPG figurative constants.

The following are VisualAge RPG-specific figurative constants. For more information, see "Figurative Constants" on page 153.

*ABORT	*BLACK	*BLUE	*BROWN
*CANCEL	*CYAN	*DARKBLUE	*DARKCYAN
*DARKGREEN	*DARKGRAY	*DARKPINK	*DARKRED
*END	*GREEN	*HALT	*IGNORE
*INFO	*NOBUTTON	*PALEGRAY	*PINK
*RED	*RETRY	*START	*YELLOW
*YESBUTTON	*WARN	*WHITE	

Data Areas

The local data area and the Program Initialization Parameters data area are not supported. You cannot use:

- The *DTAARA DEFINE operation, with *LDA or *PDA in factor 2 and a name in the result field
- DTAARA(*LDA) or DTAARA(*PDA) on a definition specification.

For more information on data area support for VisualAge RPG, see Chapter 11, "Data Structures," on page 157.

Arrays and Tables

VisualAge RPG does not support the following operation codes for arrays and tables: MLLZO, MHHZO, MLHZO, MHLZO

The AS/400 system is an EBCDIC system while the OS/2 system is an ASCII system. VisualAge RPG uses the ASCII collating sequence. For more information, see Chapter 12, "Using Arrays and Tables," on page 171.

Note: Graphic data and the ALTSEQ keyword are not supported for arrays.

For more information on VisualAge RPG arrays and tables, see Chapter 12, "Using Arrays and Tables," on page 171.

Edit Codes

User-defined edit codes are not supported. For a description of the VisualAge RPG supported edit codes and words, see Chapter 13, "Editing Numeric Fields," on page 191. For information on editing GUI parts, see *Programming with VisualAge RPG*, SC09-2449-05.

Files

In VisualAge RPG, the contents of the device-specific input/output feedback area of the file are copied to the device-specific feedback section of the INFDS only after a POST for the file. For more information, see "POST (Post)" on page 652.

Specifications

The following records are not supported by VisualAge RPG:
- File translation records
- Alternate collating sequence records

Control Specifications

For detailed information on the VisualAge RPG Control Specifications, see Chapter 16, "Control Specifications," on page 223.

Data Areas

If you do not provide information about generating and running your application in the control specifications, ILE RPG searches for a data area named RPGLEHSPEC in the library list (*LIBL). If it is not found, ILE RPG then searches for a data area named DFTLEHSPEC in QRPGLE. VisualAge RPG does not search *LIBL or QRPGLE for any data areas.

Keywords

The following keywords are not supported on the VisualAge RPG control specification:
- ACTGRP
- ALTSEQ
- BNDDIR
- DFTACTGRP
- DFTNAME
- ENBPFRCOL
- FIXNBR
- FORMSALIGN
- FTRANS
- LANGID
- OPENOPT
- OPTIMIZE
- PRFDTA
- SRTSEQ
- TEXT
- THREAD
- USRPRF

The *{NO}SRCSTMT and *{NO}DEBUGIO values on the OPTION keyword are not supported.

The parameter to the CCSID keyword must be a workstation CCSID.

The following are VisualAge RPG specific keywords for the control specifications:
- CACHE
- CACHEREFRESH
- CVTOEM
- LIBLIST
- SQLBINDFILE

- SQLDBBLOCKING
- SQLDBNAME
- SQLDTFMT
- SQLISOLATIONLVL
- SQLPACKAGENAME
- SQLPASSWORD
- SQLUSERID

For a description of the keyword support for VisualAge RPG Control Specifications, see "Positions 7-80 (Keywords)" on page 223.

File Description Specifications

For detailed information on the VisualAge RPG File Description Specifications, see Chapter 17, "File Description Specifications," on page 237.

File Support

VisualAge RPG does not support a number of files that are supported by ILE RPG. The following lists which files are not supported by VisualAge RPG, as well as which positions on the file description specification are affected.

primary files, secondary files, record address files

ILE RPG supports a file designation (position 18) for primary files, secondary files, and record address files. VisualAge RPG does not support these files.

record address files and indexed files

- VisualAge RPG only supports an entry of blank for the length of key or record address (positions 29-33).
- The record address type (position 34) for a VisualAge RPG program cannot contain A (character keys), P (packed keys), G (graphic keys), D (date keys), T (time keys) or Z (timestamp keys).
- The file organization entry (position 35) for a VisualAge RPG program cannot contain an entry of I (indexed files) or T (record address files).

WORKSTN files

- ILE RPG supports a file type (position 17) of Combined which is valid for SPECIAL and WORKSTN files. Since VisualAge RPG does not support WORKSTN files, specifying a file type of combined only applies to SPECIAL files.

Disk file processing methods

Sequential access within limits is not supported by VisualAge RPG.

RPG Cycle Related Entries

Since the RPG cycle is not supported by VisualAge RPG, the following entries are not supported:

- End of file (E)
- Sequence (A and D)
- Limits processing (L)
- Overflow processing (OA-OG, OV, or 01-99)

Keywords

The following keywords are not supported for the File description specifications for a VisualAge RPG program:

DEVID	EXTIND	FORMOFL	INDDS
KEYLOC	MAXDEV	OFLIND	PASS

PGMNAME	RAFDATA	SAVEDS	SAVEIND
SFILE	SLN		

The following are VisualAge RPG specific File Description Specification keywords:

EXTFILE(fname)
The EXTFILE keyword allows you to specify an actual local filename at run time rather than supplying the name at compile time.

PROCNAME (procname)
If you enter SPECIAL for the device entry (position 42), the module you specify as procname handles the support for the device.

RCDLEN(fieldname)
The RCDLEN keyword can be used for local DISK files.

REMOTE
If you enter DISK (position 36-42), the REMOTE keyword specifies that the disk device is on an AS/400 server.

For a description of the keyword support for VisualAge RPG File Description Specifications, see "Positions 44-80 (Keywords)" on page 243.

Definition Specifications

VisualAge RPG supports message windows. Message windows are specified on the definition specification by entering M in position 24 and a blank in position 25. For more information on message windows, see Chapter 18, "Definition Specifications," on page 255.

Keywords

The following describes any differences for the definition specification keywords between ILE RPG and VARPG.

ASCEND and DESCEND
ILE RPG uses the EBCDIC collating sequence. VisualAge RPG uses the ASCII collating sequence.

Since VisualAge RPG does not support ALTSEQ, your VisualAge RPG application cannot use an alternate sequence to check the sequence of compile-time arrays or tables.

DTAARA
VisualAge RPG does not support local data areas (*LDA) with the DTAARA keyword.

The following keywords are not supported for the definition specifications for a VisualAge RPG program:
- EXPORT
- EXTPGM
- IMPORT
- OPDESC
- OPTIONS(*NOPASS)

The following are VisualAge RPG specific keywords for the definition specifications:

BUTTON(button1:button2...)
Use the BUTTON keyword to define the buttons on a message window.

You can specify the following parameters (a maximum of three are allowed): *OK, *CANCEL, *RETRY, *ABORT, *IGNORE, *ENTER, *NOBUTTON, *YESBUTTON.

CLTPGM(program name)
The CLTPGM keyword specifies the name of the local program called by the VARPG program, using the CALLP operation.

DLL(name)
The DLL keyword, together with the LINKAGE keyword, is used to prototype a procedure that calls functions in Windows DLLs, including Windows APIs.

LINKAGE(linkage_type)
Use the LINKAGE keyword with the parameter *SERVER to specify that the program name used with the CALL operation code is located on an AS/400 server. The LINKAGE and DLL keywords can be used together to prototype a procedure that calls functions in Windows DLLs, including Windows APIs.

MSGDATA(msgdata1:msgdata2...)
Use the MSGDATA keyword to define substitution text. The parameters (msgdata1:msgdata2...) are fieldnames. VisualAge RPG converts all data to character format before displaying the message.

MSGNBR(*MSGnnnn or fieldname)
The MSGNBR keyword defines the message number. The message number can be a maximum of 4 digits in length.

MSGTEXT('message text')
The MSGTEXT keyword defines the message text. The message text is contained in single quotation mark (').

MSGTITLE('title text')
The MSGTITLE keyword specifies the title text for the message window. You can enter a string or message number enclosed in single quotation marks(').

NOWAIT
The NOWAIT keyword is used to call remote AS/400 programs that use display files.

STYLE(style_type)
The STYLE keyword is used for message window to define the icon that appears on the message window. You can specify one of the following parameters with the STYLE keyword: *INFO, *WARN, or *HALT.

For a description of the keyword support for VisualAge RPG Definition Specifications, see "Definition-Specification Keywords" on page 264.

Input Specifications

For detailed information on the VisualAge RPG File Description Specifications, see Chapter 17, "File Description Specifications," on page 237.

The following entries are not supported:
- For sequence (positions 17-18), you cannot enter a two digit number. ILE RPG supports this option which can be used to check the sequence of the input records. This support is not available for VisualAge RPG.

- For number (position 19), you cannot enter 1 (which indicates that only one record of this type can be present in the sequenced group) or N (which indicates that one or more records of this type can be present in the sequenced group).
- For option (position 20), you cannot enter O (which indicates that the record type is optional if sequence checking is specified).
- For code part (positions 29, 37, 45), you cannot enter Z (which indicates the zone portion of a character).

Built-in Functions

%GETATR and %SETATR are VARPG-specific built-in functions. For more information, see "%GETATR (Retrieve Attribute)" on page 446 and "%SETATR (Set Attribute)" on page 471.

Note: VARPG-specific built-in functions do not support 1-byte, 8-byte, and unicode values.

Operation codes

This section compares ILE RPG operation codes to VisualAge RPG operation codes. For a complete description of the VisualAge RPG operation codes, see Chapter 26, "Operation Code Details," on page 501.

Similar Operation Codes

The following operation codes exist for both ILE RPG and VisualAge RPG. However, there are differences in the way you code these operation codes or there are different results when running applications containing these operation codes. For a description of how the operation code works in for VisualAge RPG, refer to the headings listed with the operation code.

- BEGSR ("BEGSR (Begin User Subroutine)" on page 511)
- CALL ("CALL (Call an AS/400 Program)" on page 517)
- CALLB ("CALLB (Call a Function)" on page 521)
- CALLP ("CALLP (Call a Prototyped Procedure or Program)" on page 522)
- CHAIN ("CHAIN (Random Retrieval from a File)" on page 529)
- CLEAR ("CLEAR (Clear)" on page 539)
- CLOSE ("CLOSE (Close Files)" on page 542)
- COMMIT ("COMMIT (Commit)" on page 544)
- DEFINE ("DEFINE (Field Definition)" on page 548)
- DELETE ("DELETE (Delete Record)" on page 551)
- DSPLY ("DSPLY (Display Message Window)" on page 562)
- FEOD ("FEOD (Force End of Data)" on page 580)
- READ ("READ (Read a Record)" on page 653)
- WRITE ("WRITE (Create New Records)" on page 717)

Unsupported Operation Codes

The following operation codes are not supported for VisualAge RPG:

- ACQ
- DUMP
- EXFMT
- FORCE
- MHHZO
- MHLZO
- MLHZO
- MLLZO
- NEXT

- REL
- SHTDN

VisualAge RPG Specific Operation Codes

The following operation codes are unique for the VisualAge RPG language:

- BEGACT/ENDACT (Begin Action Subroutine, End Action Subroutine)
- CLSWIN (Close Window)
- DSPLY (Display Message Window)
- GETATR/SETATR (Retrieve Attribute, Set Attribute)
- READS (Read Selected)
- SHOWWIN (Load a Window)
- START/STOP (Start a Component, Stop a Component)

For a detailed description of these operation codes, see Chapter 26, "Operation Code Details," on page 501.

Note: Except for DSPLY, VARPG-specific operation codes do not support 1-byte and 8-byte signed and unsigned integer values, and unicode values.

Conversions between CCSIDs

The VARPG compiler does not support conversions between single-byte character and graphic data. Conversions are supported only between the following:

- Graphic to UCS-2 *or* UCS-2 to Graphic
- Character to UCS-2 *or* UCS-2 to Character
- Graphic to Graphic (when their CCSIDs are different)

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM

8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents General-Use Programming Interface and Associated Guidance Information provided by IBM WebSphere Development Studio Client for iSeries.

This publication is intended to help you to create VisualAge RPG applications using RPG IV source. This publication documents General-Use Programming Interface and Associated Guidance Information provided by the VisualAge RPG compiler.

General-Use programming interfaces allow the customer to write programs that obtain the services of the VisualAge RPG compiler.

Trademarks and Service Marks

The following terms are trademarks or registered trademarks of the International Business Machines Corporation in the United States or other countries or both:

Application System/400	AS/400	Common User Access
CUA	DATABASE 2	DB2
DB2/400	DB2/6000	IBM
Integrated Language Environment	ISeries	OS/400
PROFS	SQL/DS	SQL/400
VisualAge	WebSphere	400

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

ActiveX, Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary includes terms and definitions from:

- The *American National Dictionary for Information Systems* ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York, 10018. Definitions are defined by the symbol (A) after the definition.
- The *Information Technology Vocabulary* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Committee (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition indicating that the final agreement has not yet been reached among participating National Bodies of SC1.
- *IBM Dictionary of Computing* , New York: McGraw-Hill, 1994.
- *Object-Oriented Interface Design IBM Common User Interface Guidelines*, SC34-4399-00, Carmel, IN: Que Corporation, 1992.

A

action. (1) Synonym for *action subroutine*. (2) An executable program or command file used to manipulate a project's parts or participate in a build.

action subroutine. Logic that you write to respond to a specific event.

active window. The window with which a user is currently interacting. This is the window that receives keyboard input.

activeX part. A part that adds ActiveX control objects to the project. VARPG applications can then access their attributes and monitor for events.

anchor. Any part that you use as a reference point for aligning, sizing, and spacing other parts.

animation control part. A part that allows the playback of video files, with the AVI extension, in Windows, or the playback of animated GIF sequences in Java applications.

API. Application programming interface.

applet. A program that is written in Java and runs inside of a Java-compatible browser or AppletViewer.

application. A collection of software components used to perform specific user tasks on a computer.

application programming interface (API). A functional interface supplied by the operating system or a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

B

BMP. The file extension of a bitmap file.

build. The process by which the various pieces of source code that make up components of a VARPG application are compiled and linked to produce an executable version of the application.

button. (1) A mechanism on a pointing device, such as a mouse, used to request or start an action. (2) A graphical mechanism in a window that, when selected, results in an action. An example of a button is an OK push button that, when selected, initiates an action.

C

calendar part. A part that adds a calendar that can be modified by the user to include text, color and other attributes.

canvas part. A part onto which you can point and click various other parts, position them, and organize them to produce a graphical user interface. A canvas part occupies the client area of either a window part or a notebook page part. See also *notebook page with canvas part* and *window with canvas part*.

check box part. A square box with associated text that represents a choice. When a user selects a choice, an indicator appears in the check box to indicate that the choice is selected. The user can clear the check box by selecting the choice again. In VisualAge RPG, you point and click on a check box part in the parts palette or parts catalog and click it onto a design window.

click. To press and release a mouse button without moving the pointer off of the choice or object. See also *double-click*.

client. (1) A system that is dependent on a server to provide it with data. (2) The PWS on which the VARPG applications run. See also *DDE client*.

client area. The portion of the window that is the user's workspace, where a user types information and selects choices from selection fields. In primary windows, the area where an application programmer presents the objects that a user works on.

client/server. The model of interaction in distributed data processing in which a program at one site sends a request to a program at another site and awaits a response. The requesting program is called a client; the answering program is called a server. See also *client*, *server*, *DDE client*, *DDE server*.

clipboard. An area of storage provided by the system to hold data temporarily. Data in the clipboard is available to other applications.

cold-link conversation. In DDE, an explicit request made from a client program to a server program. The server program responds to the request. Contrast with *hot-link conversation*.

color palette. A set of colors that can be used to change the color of any part in your application's GUI.

combination box. A control that combines the functions of an entry field and a list box. A combination box contains a list of objects that a user can scroll through and select from to complete the entry field. Alternatively, a user can type text directly into the entry field. In VisualAge RPG, you can point and click on a combination box part in the parts palette or parts catalog and click it onto a design window.

Common User Access architecture (CUA architecture). Guidelines for the dialog between a human and a workstation or terminal.

compile. To translate a source program into an executable program (an object program).

component. A functional grouping of related files within a project. A component is created when the NOMAIN and EXE keywords are not present on the control specifications.

component reference part. A part that enables one component to communicate with another component in a VARPG application.

***component part.** A part that is the "part representation" of the component. One *component part is created for each component automatically, and it is invisible.

CONFIG.SYS. The configuration file, located in the root directory of the boot drive, for the DOS, OS/2, or Windows operating systems. It contains information required to install and run hardware and software.

configuration. The manner in which the hardware and software of an information processing system are organized and interconnected (T).

container part. A part that stores related records and displays them in a details, icon, or tree view.

CUA architecture. Common User Access architecture.

cursor. The visible indication of the position where user interaction with the keyboard will appear.

D

database. (1) A collection of data with a given structure for accepting, storing, and providing, on demand, data for multiple users. (T) (2) All the data files stored in the system.

data object. An object that conveys information, such as text, graphics, audio, or video.

DBCS. Double-byte character set.

DDE. Dynamic data exchange.

DDE client. An application that initiates a DDE conversation. Contrast with *DDE server*. See also *DDE client part*, *DDE conversation*.

DDE client part. A part used to exchange data with other applications, such as spreadsheet applications, that support the dynamic data exchange (DDE) protocol.

DDE conversation. The exchange of data between a DDE client and a DDE server. See also *cold-link conversation* and *hot-link conversation*.

DDE server. An application that provides data to another DDE-enabled application. Contrast with *DDE client*. See also *DDE conversation*.

default. A value that is automatically supplied or assumed by the system or program when no value is specified by the user. The default value can be assigned to a push button or graphic push button.

default action. An action that will be performed when some action is taken, such a pressing the Enter key.

dereferencing. The action of removing the association between a part and an AS/400 database field.

design window. The window in the GUI designer on which parts are placed to create a user interface.

details view. A standard contents view in which a small icon is combined with text to provide descriptive information about an object.

dimmed. Pertaining to the reduced contrast indicating that a part can not be selected or directly manipulated by the user.

direct editing. The use of techniques that allow a user to work with an object by dragging it with a mouse or interacting with its pop-up menu.

DLL. Dynamic link library.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with *single-byte character set (SBCS)*.

double-click. To quickly press a mouse button twice.

drag. To use a mouse to move or to copy an object. For example, a user can drag a window border to make it larger by holding a button while moving the mouse. See also *drag and drop*.

drag and drop. To directly manipulate an object by moving it and placing it somewhere else using a mouse.

drop-down combination box. A variation of a combination box in which a list box is hidden until a user takes explicit acts to make it visible.

drop-down list. A single selection field in which only the current choice is visible. Other choices are hidden until the user explicitly acts to display the list box that contains the other choices.

dynamic data exchange (DDE). The exchange of data between programs or between a program and a datafile object. Any change made to information in one program or session is applied to the identical data created by the other program. See also *DDE conversation*, *DDE client*, *DDE server*.

Dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

E

EBCDIC. Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters.

emphasis. Highlighting, color change, or other visible indication of conditions relative to an object or choice that affects a user's ability to interact with that object or choice. Emphasis can also give a user additional information about the state of a choice or an object.

entry field part. An area on a display where a user can enter information, unless the field is read-only. The boundaries of an entry field are usually indicated. In VisualAge RPG, you point and click on an entry field part in the parts palette or parts catalog and click it onto a design window.

error logging. Keeps track of errors in an **error log**. The editor takes you to the place in the source where the error occurred.

event. A signal generated as a result of a change to the state of a part. For example, pressing a button generates a *Press* event.

exception. (1) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it. (I) (2) In VisualAge RPG, an event or situation that prevents, or could prevent, an action requested by a user from being completed in a manner that the user would expect. Exceptions occur when a product is unable to interpret a user's input.

EXE. The extension of an executable file.

EXE module. An EXE module consists of a main procedure and subprocedures. It is created when the EXE keyword is present on the control specification. All subroutines (BEGSR) must be local to a procedure. The EXE must contain a procedure whose name matches the name of the source file. This will be the main entry point for the EXE, that is, the main procedure.

export. A function that converts an internal file to some standard file format for use outside of an application. Contrast with *import*.

F

field. (1) An identifiable area in a window, such as an entry field where a user types text. (2) A group of related bytes, such as a name or amount, that is treated as a unit in a record.

file. A collection of related data that is stored and retrieved by an assigned name. A file can include information that starts a program (program-file object), contains text or graphics (data-file object), or processes a series of commands (batch file).

focus. Synonym for *input focus*.

font palette. A set of fonts that can be used to change the font of a part in your application's GUI.

G

graph part. A part that allows the user to add a graph to the GUI. The graph styles available are line, bar, line and bar, or pie chart.

graphical user interface (GUI). A type of user interface that takes advantage of high-resolution graphics. A graphical user interface includes a combination of graphics, the object-action paradigm, the use of pointing devices, menu bars and other menus, overlapping windows, and icons.

graphic push button part. A push button, labeled with a graphic, that represents an action that will be initiated when a user selects it. Contrast with *push button part*.

group box part. A rectangular frame around a group of controls to indicate that they are related and to provide an optional label for the group. In VisualAge RPG, you point and click on a group box part in the parts palette or parts catalog and click it onto a design window.

group marker. A mark that identifies a part as being the first one in a group. When a user moves the cursor through a group of parts and reaches the last part, the cursor returns to the first part in the group.

GUI designer. A suite of tools used to create interfaces by dragging and dropping parts from the parts palette to the design window.

H

hide button. A button on a title bar that a user clicks on to remove a window from the workplace without closing the window. When the window is hidden, the state of the window, as represented in the window list, changes. Contrast with *maximize button* and *minimize button*.

horizontal scroll bar part. A part that adds a horizontal scroll bar to a window. This part allows users to scroll through a pane of information, from left-to-right or right-to-left.

hot-link conversation. In DDE, an automatic update of a client program by a server program when data changes on the server. Contrast with *cold-link conversation*.

I

ICO. The file extension of an icon file.

icon. A graphical representation of an object, consisting of an image, image background, and a label.

icon view. A standard contents view in which each object contained in a container is displayed as an icon.

image part. A part used to display a picture, from a BMP or ICO file, on a window.

import. A function that converts AS/400 display file objects to the appropriate VARPG part. Contrast with *export*.

inactive window. A window that can not receive keyboard input at a given moment.

index. The identifier of an entry in VARPG parts such as list boxes or combination boxes.

information area. A part of a window in which information about the object or choice that the cursor is on is displayed. The information area can also contain a message about the normal completion of a process. See also *status bar*.

Information Presentation Facility (IPF). A tool used to create online help on a programmable workstation.

Information Presentation Facility (IPF) file. A file in which the application's help source is stored.

INI. The file extension for a file in the OS/2 or Windows operating system containing application-specific information that needs to be preserved from one call of an application to another.

input focus. The area of a window where user interaction is possible from either the keyboard or the mouse.

input/output (I/O). Data provided to the computer or data resulting from computer processing.

IPF. Information Presentation Facility

item. In dynamic data exchange, a unit of data. For example, the top left cell position in a spreadsheet is row 1, column 1. This cell position may be referred to as item R1C1.

J

JAR files (.jar). In Java, abbreviation for Java ARchive. A file format that is used for aggregating many files into one.

Java. An object-oriented programming language for portable interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated.

java bean part. A part that allows VARPG applications to access Sun Microsystem's JavaBeans.

JavaBeans. In Java, a portable, platform-independent reusable component model.

Java Database Connectivity (JDBC). An industry standard for database-independent connectivity between Java and a wide range of databases. The JDBC provides a call-level application programming interface (API) for SQL-based database access.

Java 2 Software Development Kit (J2SDK). Software that Sun Microsystems distributes for Java developers. This software includes the Java interpreter, Java classes, and Java development tools. The development tools include a compiler, debugger, dissassembler, AppletViewer, stub file generator, and documentation generator.

Java Native Interface (JNI). A programming interface that allows Java code that runs inside of a Java Virtual Machine (JVM) to interoperate with functions that are written in other programming languages.

Java Runtime Environment (JRE). A subset of the Java Developer Kit for end users and developers who want to redistribute the JRE. The JRE consists of the Java Virtual Machine, the Java Core Classes, and supporting files.

Java Virtual Machine (JVM). The part of the Java Runtime Environment (JRE) that is responsible for interpreting Java bytecodes.

L

link event. An event that a target part receives whenever the state of a source part changes.

list box part. A control that contains scrollable choices that a user can select. In VisualAge RPG, you can point and click on a list box part in the parts palette or parts catalog and click it onto a design window.

M

main procedure. A main procedure is a subprocedure that can be specified as the program entry procedure and receives control when it is first called. A main procedure is only produced when creating an EXE. See *EXE module*

main source section. In a VARPG program, the main source section contains all the global dedfinitions for a module. For a component, this section also includes the action and user subroutines.

main window. See *primary window*.

manipulation button. See *mouse button 2*.

maximize button. A button on the rightmost part of a title bar that a user clicks on to enlarge the window to its largest possible size. Contrast with *minimize button*, *hide button*.

media panel part. A part used to give the user control over other parts. For example, a media panel part can be used to control the volume of a media part.

media part. A part that gives a program the ability to process sound files and video files.

menu. A list of choices that can be applied to an object. A menu can contain choices that are not available for selection in certain contexts. Those choices are dimmed.

menu bar part. The area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus. In VisualAge RPG, you can point and click on a menu bar part in the parts palette or parts catalog and click it onto a design window.

menu item part. A part that is a graphical or textual item on a menu. A user selects a menu item to work with an object in some way.

message. (1) Information not requested by a user but displayed by a product in response to an unexpected event or when something undesirable could occur. (2) A communication sent from a person or program to another person or program.

message file. A file containing application messages. The file is created from the message source file during the build process. See also *build*.

message subfile part. A part that can display predefined messages or text supplied in program logic.

migrate. (1) To move to a changed operating environment, usually to a new release or version of a system. (2) To move data from one hierarchy of storage to another.

MID. The file extension of a MIDI file.

MIDI file. Musical Instrument Digital Interface file.

minimize button. A button, located next to the rightmost button in a title bar, that reduces the window to its smallest possible size. Contrast with *maximize button* and *hide button*.

mnemonic. A single character, within the text of a choice, identified by an underscore beneath the character. See also *mnemonic selection*.

mnemonic selection. A selection technique whereby a user selects a choice by typing the mnemonic for that choice.

mouse. A device with one or more push buttons used to position a pointer on the display without using the keyboard. Used to select a choice or function to be performed or to perform operations on the display, such as dragging or drawing lines from one position to another.

mouse button. A mechanism on a mouse used to select choices, initiate actions, or manipulate objects with the pointer. See also *mouse button 1* and *mouse button 2*.

mouse button 1. By default, the left button on a mouse used for selection.

mouse button 2. By default, the right button on a mouse used for manipulation.

mouse pointer. Synonym for *cursor*.

multiline edit (MLE) part. A part representing an entry field that allows the user to enter multiple lines of text.

N

navigation panel. A group of buttons that can be used to control the visible selection of records in a subfile.

NOMAIN module. A module that contains only subprocedures. There are no action or standalone user subroutines in it. A NOMAIN module is created when the NOMAIN keyword is present on the control specification.

notebook part. A graphical representation of a notebook. You can add notebook pages to the notebook part and then group the pages into sections separated by tabbed dividers. In Windows, a notebook is sometimes referred to as a Windows tab control. See also *notebook page part*, *notebook page with canvas part*.

notebook page part. A part used to add pages to a notebook part. See also *notebook*.

notebook page with canvas part. A combination of a notebook page part and a canvas page part. See also *notebook*, *canvas part*.

O

object. (1) A named storage space that consists of a set of characteristics that describe itself and, in some situations, data. An object is anything that exists in and occupies space in storage and on which operations can be performed. Some examples of objects are programs, files, libraries, and folders. (2) A visual component of a user interface that a user can work with to perform a task. An object can appear as text or an icon.

object-action paradigm. A pattern for interaction in which a user selects an object and then selects an action to apply to that object.

object-oriented programming. A method for structuring programs as hierarchically organized classes describing the data and operations of objects that may interact with other objects. (T)

object program. A target program suitable for execution. An object program may or may not require linking. (T)

odbc/jdbc part. A part that allows VAPRG applications to access and process database files that support the Windows ODBC API or Sun Microsystem's JDBC API.

operating system. A collection of system programs that control the overall operation of a computer system.

outline box part. A part that is a rectangular box positioned around a group of parts to indicate that all the parts are related.

P

package. A function used to collect all the parts of a VARPG application together for distribution.

parts. Objects that make up the GUI of a VARPG application.

parts catalog. A storage space for all of the parts used to create graphical user interfaces for VARPG applications.

parts palette. A collection of parts that are most appropriate for building the current graphical user interface for an application. When you finish one GUI, you can wipe the palette clean and add parts from the parts catalog that you require for the next application.

plugin. A function created by the user or an outside vendor that can be used in VARPG programs.

point and click. (1) A selection method which is used to copy a part from the parts palette or catalog to the GUI design window, the icon view, or the tree view. (2) To place a part in any of the desired views, point to and click on the part, then move the cursor to the chosen window and point the cursor and click where you want the part to appear. In the icon and tree views, the part will be placed on the parent part, and you will then have to move it where you would like it to appear in the design window.

pop-up menu. A menu that, when requested, appears next to the object with which it is associated. It contains choices appropriate for the object in its current context.

pop-up menu part. A part that, when added to an object on your interface, appears next to the object with which it is associated when requested. You can point and click on a pop-up menu part in the parts palette or parts catalog and click it onto a design window.

pop-up window. A movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request. Synonymous with *secondary window*.

primary window. The window in which the main interaction between the user and the application takes place. Synonymous with *main window*.

procedure. A procedure is any piece of code that can be called with the CALLP operation code.

procedure interface definition. A procedure interface definition is a repetition of the prototype information within the definition of a procedure. It is used to declare the entry parameters for the procedure and to ensure that the internal definition of the procedure is consistent with the external definition (the prototype)

programmable workstation (PWS). A workstation that has some degree of processing capability and that allows a user to change its functions.

progress bar part. A part that can be used to indicate graphically the progress of a process, such as copying files, loading a database, and so on.

progress indicator. One or more controls used to inform a user about the progress of a process.

project. The complete set of data and actions needed to build a single target, such as dynamic link library (DLL) or an executable file (EXE).

prompt. (1) A visual or audible message sent by a program to request the user's response. (T) (2) A displayed symbol or message that requests input from the user or gives operational information. The user must respond to the prompt in order to proceed.

properties notebook. A graphical representation that resembles a bound notebook containing pages separated into sections by tabbed divider pages. Select the tabs of a notebook to move from one section to another.

prototype. A prototype is a definition of the call interface. It includes information such as: whether the call is bound (procedure) or dynamic (program); the external name; the number and nature of the parameters; which parameters must be passed; the data type of any return value (for a procedure)

pull-down menu. A menu that extends from a selected choice on a menu bar or from a system-menu symbol. The choices in a pull-down menu are related to one another in some manner.

push button part. A button labeled with text that represents an action that starts when a user selects the push button. You can point and click on a push button part in the parts palette or parts catalog and click it onto a design window. See also *graphic push button part*.

PWS. Programmable workstation.

R

radio button part. A circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected. You can point and click on a radio button part in the parts palette or parts catalog and click it onto a design window.

reference field. An AS/400 database field from which an entry field part can inherit its characteristics.

restore button. A button that appears in the rightmost corner of the title bar after a window has been maximized. When the restore button is selected, the window returns to the size and position it was in before it was maximized. See also *maximize button*.

S

SBCS. Single-byte character set.

scroll bar. A part that shows a user that more information is available in a particular direction and can be moved into view by using a mouse or the page keys.

secondary window. A window that contains information that is dependent on information in a primary window, and is used to supplement the interaction in the primary window. See also *primary window*. Synonym for *pop-up window*.

secure sockets layer (SSL). A popular security scheme that was developed by Netscape Communications Corp. and RSA Data Security, Inc. SSL allows the client to authenticate the server and all data and requests to be encrypted. The URL of a secure server that is protected by SSL begins with https rather than http.

selection border. The visual border that appears around a VARPG part or a custom-made part, allowing the part to be moved with the mouse or keyboard.

selection button. See *mouse button 1*.

server. A system in a network that handles the requests of another system, called a client.

server alias. A name you define that can be used instead of the server name.

shared component. A component that can be accessed by more than one project.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set (DBCS)*.

sizing border. The border or frame around a part (or set of parts) that you select to resize the part (or set of parts) using the mouse or the keyboard.

slider part. A visual component of a user interface that represents a quantity and its relationship to the range of possible values for that quantity. A user can also change the value of the quantity. You can point and click on a slider part in the parts palette or parts catalog and click it onto a design window.

slider arm. The visual indicator in the slider that a user can move to change the numerical value.

source directory. The directory in which all source files for a VARPG application are stored.

source part. A part that can notify target parts whenever the state of the source part changes. A source part can have multiple targets.

spin button part. A type of entry field that shows a ring of related but mutually exclusive choices through which a user can scroll and select one choice. A user can also type a valid choice in the entry field. You can point and click on a spin button part in the parts palette or parts catalog and click it onto a design window.

SSL. Secure sockets layer.

static text part. A part used as a label for other parts, such as a prompt for an entry field part.

status bar. A part of a window that displays information indicating the state of the current view or object. See also *information area*.

status bar part. A part on a window that can display additional information about a process or action for the window.

subfile field. A field used to define fields in a subfile part. See also *subfile part*.

subfile part. A part used to display a list of records, each consisting of a number of fields. This part is similar to an AS/400 subfile. See also *subfile field*.

submenu. A menu that appears from, and contains choices related to, a cascading choice in another menu. Submenus are used to reduce the length of a pull-down menu or a pop-up menu. See also *submenu part*.

submenu part. A part used to start a submenu from a menu item or existing menu, or to start a pull-down menu from a menu item on a menu bar. See also *submenu* and *menu item part*.

subprocedure. A subprocedure is a procedure specified after the main source section. It must have a corresponding prototype in the definition specifications of the main source section

syntax checking. Verifies that the syntax of each line is correct while you are editing the source. By doing so, it can avoid compile errors. You can set this option on or off. You can view only certain specification types, such as C specs, or a line with a specific string.

T

tab stop. An attribute used to set a tab stop for a part so that users can focus on it when they use the Tab key to move through the interface.

target part. A part that receives a link event from a source part whenever the state of the source part changes.

target directory. The directory in which the compiled VARPG application is stored after a build. Contrast with *target folder*.

target folder. The object in which the icon representing a VARPG application is placed.

target program. The object to be built by the project, such as a dynamic link library (DLL).

thread. The smallest unit of operation to be performed within a process.

timer part. A part used to track the interval of time between two events and trigger the second event when the interval has passed.

title bar. The area at the top of each window that contains the system-menu symbol.

token highlighting. Enhances the readability of the code. You can configure highlighting of different language constructs with different colors or fonts to identify the program structures. You can turn token highlighting on or off.

tool bar. A menu that contains one or more graphical choices representing actions a user can perform using a mouse.

topic. In dynamic data exchange (DDE), the set of data that is the subject of a DDE conversation.

tree view. A way of displaying the contents of an object in a hierarchical fashion.

U

user-defined part. A part, consisting of one or more parts you have customized, that you save to the parts palette or parts catalog for reuse. When in the palette or catalog, you can point and click this part onto the design window as you would any other VARPG part.

utility DLL. See *NOMAIN module*

V

vertical scroll bar part. A part that adds a vertical scroll bar to a window. This part allows users to scroll through a pane of information vertically.

W

WAV. The file extension of a wave file.

wave file. A file used for audio sounds on a waveform device.

window part. An area with visible boundaries that represents a view of an object or with which a user conducts a dialog with a computer system. You can point and click on a window part from the parts palette or parts catalog and click it onto the project window.

window with canvas part. A combination of the window part and the canvas part. See also *window part* and *canvas part*.

work area. An area used to organize objects according to a user's tasks. When a user closes a work area, all windows opened from objects contained in the work area are removed from the workplace.

workplace. An area that fills the entire display and holds all of the objects that make up the user interface.

workstation. A device that allows a user to do work. See also *programmable workstation*.

Bibliography

For additional information about topics related to WebSphere Development Studio Client, refer to the following IBM publications:

WebSphere Development Studio Client manuals:
- *Getting Started with WebSphere Development Studio Client for iSeries*, SC09-2625-06, provides information about WebSphere Development Studio Client for iSeries, giving an overview of the various components, how they work together, and the business advantages of using them.

VisualAge RPG manuals:
- *Programming with VisualAge RPG*, SC09-2449-05, contains specific information about creating applications with VisualAge RPG. It describes the steps you have to follow at every stage of the application development cycle, from design to packaging and distribution. Programming examples are included to clarify the concepts and the process of developing VARPG applications.
- *VisualAge RPG Parts Reference*, SC09-2450-05, provides a description of each VARPG part, part attribute, part event, part attribute, and event attribute. It is a reference for anyone who is developing applications using VisualAge RPG.
- *VisualAge RPG Language Reference*, SC09-2451-04, provides reference information about the VARPG language and compiler.
- *Java for RPG Programmers* introduces you to the Java language (and RPG IV) by comparing it to the RPG language. It is a good first step in your Java journey. It also includes an interactive CD tutorial on Java and VisualAge for Java, by MINDQ.
- *Experience RPG IV Tutorial* is an interactive CD tutorial that teaches you RPG IV and ILE, in a fun and step-by-step approach. The book is a handbook with questions and exercises to help you get hands-on experience with this exciting new version of RPG.
- Another non-IBM book of interest to VisualAge RPG users is *VisualAge for RPG by Example*.

If you have internet access, you can obtain current iSeries and AS/400e information and publications from the following Web site:

`http://www.ibm.com/eserver/iseries/infocenter`

For the PDF version of iSeries publications, refer to the CD ROM *iSeries Information Center: Supplemental Manuals*, SK3T-4092-00.

Application Development Manager manuals:
- *ADTS/400: Application Development Manager Introduction and Planning Guide*, GC09-1807-00, describes the basic concepts and the planning needed to make effective use of the Application Development Manager feature.
- *ADTS: Application Development Manager User's Guide*, SC09-2133-02, describes how to create and manage projects defined to the Application Development Manager feature.
- *ADTS/400: Application Development Manager Self-Study Guide*, SC09-2138-00, provides practical hands-on experience using the Application Development Manager feature. The guide illustrates how to use the Application Development Manager feature by leading you through a series of step-by-step exercises.

- *ADTS/400: Application Development Manager API Reference*, SC09-2180-00, describes how application programmers can write their own interface to the Application Development Manager feature.

Information Presentation Facility manual:
- *Information Presentation Facility Programming Guide* G25H-7110, describes the elements that make up the Information Presentation Facility (IPF). IPF is a tool that supports the design and development of online documents and online help facilities.

SQL manuals:
- *IBM SQL Reference Version 2* SC26-8416, Volume 2, compares the facilities of
  - DB2
  - SQL/DS™
  - DB2/400™
  - DB2/6000™
  - IBM SQL
  - ISO-ANSI (SQL92E)
  - X/Open™ (XPG4-SQL).
- *DB2 Universal Database Administration Guide* S10J-8157, provides information necessary to use and administer the DB2 product.
- *DB2 Universal Database Embedded SQL Programming Guide* S10J-8158, describes how to design and code application programs that access DB2 Client/Server family servers (such as DB2 or DB2/400). It presents detailed information on the use of Structured Query Language (SQL), and API calls in applications.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Printed in U.S.A.

SC09-2451-06