IBM

Rational Development Studio for i

ILE RPG Language Reference

7.1

SC09-2508-08

Rational Development Studio for i

ILE RPG Language Reference

7.1

SC09-2508-08

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 915.

| This edition applies to Version 7, Release 1, Modification Level 0, of IBM® Rational® Development Studio for ILE
| RPG Programmer's Guide (5770-WDS), and to all subsequent releases and modifications until otherwise indicated in
| new editions. This edition applies only to reduced instruction set computer (RISC) systems.

| This edition replaces SC09-2508-07.

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

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

You can also send your comments by FAX (attention: RCF Coordinator), or you can send your comments electronically to IBM. See "How to Send Your Comments" 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

Chapter 14. Definition Specifications 315

Chapter 15. Input Specifications . . . 375

Chapter 20. Expressions 477

Chapter 21. Built-in Functions 493

About This Reference

This reference provides information about the RPG IV language as it is implemented using the ILE RPG compiler with the IBM i® (IBM i®) operating system, formerly Operating System/400® (OS/400®).

This reference covers:

- Basics of RPG IV:
  - RPG IV character set
  - RPG IV reserved words
  - Compiler directives
  - RPG IV program cycle
  - Indicators
  - Error Handling
  - Subprocedures
- Definitions:
  - Defining Data and Prototypes
  - Data types and Data formats
- RPG IV specifications:
  - Control
  - File description
  - Definition
  - Input
  - Calculation
  - Output
  - Procedure
- Ways to manipulate data or devices:
  - Built-in Functions
  - Expressions
  - Operation Codes

Who Should Use This Reference

This reference is for programmers who are familiar with the RPG IV programming language.

This reference provides a detailed description of the RPG IV language. It does not provide information on how to use the ILE RPG compiler or how to convert RPG III programs to ILE RPG. For information on those subjects, see the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*, SC09-2507-08.

Before using this reference, you should

- Know how to use applicable IBM i menus and displays or Control Language (CL) commands.
- Have a firm understanding of Integrated Language Environment® as described in detail in the *ILE Concepts*, SC41-5606-09.

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/systems/i/infocenter/`
- From CD-ROMs that ship with your Operating System/400 order:

  i5/OS Information Center CD, SK3T-4091.

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.

For a list of related publications, see the "Bibliography" on page 913.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other iSeries documentation.

- If you prefer to send comments by mail, use the the following address:

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

  If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use this number: 1–845–491–7727
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

    RCHCLERK@us.ibm.com
  - Comments on the iSeries Information Center:

    RCHINFOC@us.ibm.com

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.

What's New

There have been several releases of RPG IV since the first V3R1 release. The following is a list of enhancements made for each release since V3R1 to the current release:

You can use this section to link to and learn about new RPG IV functions.

Note: The information for this product is up-to-date with the V7R1 release of RPG IV. If you are using a previous release of the compiler, you will need to determine what functions are supported on your system. For example, if you are using a V5R1 system, the functions new to the V7R1 release will not be supported.

What's New in this Release

This section describes the enhancements made to ILE RPG in V7R1.

Sort and search data structure arrays

Data structure arrays can be sorted and searched using one of the subfields as a key.

```
// Sort the custDs array by the amount_owing subfield
SORTA custDs(*).amount_owing;

// Search for an element in the custDs array where the
// account_status subfield is "K"
elem = %LOOKUP("K" : custDs(*).account_status);
```

Sort an array either ascending or descending

An array can be sorted ascending using SORTA(A) and descending using SORTA(D). The array cannot be a sequenced array (ASCEND or DESCEND keyword).

```
// Sort the salary array in descending order
SORTA(D) salary;
```

New built-in function %SCANRPL (scan and replace)

The %SCANRPL built-in function scans for all occurrences of a value within a string and replaces them with another value.

```
// Replace NAME with 'Tom'
string1 = 'See NAME. See NAME run. Run NAME run.';
string2 = %ScanRpl('NAME' : 'Tom' : string1);
// string2 = 'See Tom. See Tom run. Run Tom run.'
```

%LEN(varying : *MAX)

The %LEN builtin function can be used to obtain the maximum number of characters for a varying-length character, UCS-2 or Graphic field.

Use ALIAS names in externally-described data structures

Use the ALIAS keyword on a Definition specification to indicate that you want to use the alternate names for the subfields of externally-described data structures. Use the ALIAS keyword on a File specification to indicate that you want to use the alternate names for LIKEREC data structures defined from the records of the file.

```
A          R CUSTREC
A            CUSTNM        25A        ALIAS(CUSTOMER_NAME)
A            CUSTAD        25A        ALIAS(CUSTOMER_ADDRESS)
A            ID            10P 0

D custDs       e ds                  ALIAS
D                                    QUALIFIED EXTNAME(custFile)
 /free
   custDs.customer_name = 'John Smith';
   custDs.customer_address = '123 Mockingbird Lane';
   custDs.id = 12345;
```

Faster return values

A procedure defined with the RTNPARM keyword handles the return value as a hidden parameter. When a procedure is prototyped to return a very large value, especially a very large varying value, the performance for calling the procedure can be significantly improved by defining the procedure with the RTNPARM keyword.

```
D getFileData     pr             a   varying len(1000000)
D                                    rtnparm
D   file                         a   const varying len(500)
D data            S              a   varying len(1000)
 /free
    data = getFileData ('/home/mydir/myfile.txt');
```

%PARMNUM built-in function

The %PARMNUM(parameter_name) built-in function returns the ordinal number of the parameter within the parameter list. It is especially important to use this built-in function when a procedure is coded with the RTNPARM keyword.

```
D                 pi
D   name                      100a   const varying
D   id                         10i 0 value
D   errorInfo                        likeds(errs_t)
D                                    options(*nopass)
 /free
   // Check if the "errorInfo" parameter was passed
   if %parms >= %parmnum(errorInfo);
```

Optional prototypes

If a program or procedure is not called by another RPG module, it is optional to specify the prototype. The prototype may be omitted for the following types of programs and procedures:

- A program that is only intended to be used as an exit program or as the command-processing program for a command
- A program that is only intended to be called from a different programming language
- A procedure that is not exported from the module
- A procedure that is exported from the module but only intended to be called from a different programming language

Pass any type of string parameter

Implicit conversion will be done for string parameters passed by value or by read-only reference. For example, a procedure can be prototyped to have a CONST UCS-2 parameter, and character expression can be passed as a parameter on a call to the procedure. This enables you to write a single procedure with the parameters and return value prototyped with the UCS-2 type. To call that procedure, you can pass any type of string parameter, and assign the return value to any type of string variable.

```
// The makeTitle procedure upper-cases the value
// and centers it within the provided length
alphaTitle = makeTitle(alphaValue : 50);
ucs2Title = makeTitle(ucs2Value : 50);
dbcsTitle = makeTitle(dbcsValue : 50);
```

Two new options for XML-INTO

- The *datasubf* option allows you to name a subfield that will receive the text data for an XML element that also has attributes.
- The *countprefix* option reduces the need for you to specify the *allowmissing=yes* option. It specifies the prefix for the names of the additional subfields that receive the number of RPG array elements or non-array subfields set by the XML-INTO operation.

These options are also available through a PTF for V6R1.

Teraspace storage model

RPG modules and programs can be created to use the teraspace storage model or to inherit the storage model of their caller. With the teraspace storage model, the system limits regarding automatic storage are significantly higher that for the single-level storage model. There are limits for the amount of automatic storage for a single procedure and for the total automatic storage of all the procedures on the call stack.

Use the storage model (STGMDL) parameter on the CRTRPGMOD or CRTBNDRPG command, or use the STGMDL keyword on the Control specification.

`*TERASPACE`
The program or module uses the teraspace storage model.

`*SNGLVL`
The program or module uses the single-level storage model.

`*INHERIT`
The program or module inherits the storage model of its caller.

Change to the ACTGRP parameter of the CRTBNDRPG command and the ACTGRP keyword on the Control specification

The default value of the ACTGRP parameter and keyword is changed from QILE to *STGMDL.

ACTGRP(*STGMDL) specifies that the activation group depends on the storage model of the program. When the storage model is *TERASPACE, ACTGRP(*STGMDL) is the same as ACTGRP(QILETS). Otherwise, ACTGRP(*STGMDL) is the same as ACTGRP(QILE).

Note: The change to the ACTGRP parameter and keyword does not affect the default way the activation group is assigned to the program. The default value for the STGMDL parameter and keyword is *SNGLVL, so when the ACTGRP parameter or keyword is not specified, the activation group of the program will default to QILE as it did in prior releases.

Allocate teraspace storage

Use the ALLOC keyword on the Control specification to specify whether the RPG storage-management operations in the module will use teraspace storage or single-level storage. The maximum size of a teraspace storage allocation is significantly larger than the maximum size of a single-level storage allocation.

Encrypted listing debug view

When a module's listing debug view is encrypted, the listing view can only be viewed during a debug session when the person doing the debugging knows the encryption key. This enables you to send debuggable programs to your customers without enabling your customers to see your source code through the listing view. Use the DBGENCKEY parameter on the CRTRPGMOD, CRTBNDRPG, or CRTSQLRPGI command.

Table 1. Changed Language Elements Since V6R1

<table>
<tr><th>Language Unit</th><th>Element</th><th>Description</th></tr>
<tr><td>Control specification keywords</td><td>ACTGRP(*STGMDL)</td><td>*STGMDL is the new default for the ACTGRP keyword and command parameter. If the program uses the teraspace storage module, the activation group is QILETS. Otherwise it is QILE.</td></tr>
<tr><td>Built-in functions</td><td>%LEN(varying-field : *MAX)</td><td>Can now be used to obtain the maximum number of characters of a varying-length field.</td></tr>
<tr><td>Operation codes</td><td>SORTA(A | D)</td><td>The SORTA operation code now allows the A and D operation extenders indicating whether the array should be sorted ascending (A) or descending (D).</td></tr>
</table>

Table 2. New Language Elements Since V6R1

<table>
<tr><th>Language Unit</th><th>Element</th><th>Description</th></tr>
<tr><td rowspan="2">Control specification keywords</td><td>STGMDL(*INHERIT | *TERASPACE | *SNGLVL)</td><td>Controls the storage model of the module or program</td></tr>
<tr><td>ALLOC(*STGMDL | *TERASPACE | *SNGLVL)</td><td>Controls the storage model for the storage-managent operations %ALLOC, %REALLOC, DEALLOC, ALLOC, REALLOC</td></tr>
<tr><td>File specification keywords</td><td>ALIAS</td><td>Use the alternate field names for the subfields of data structures defined with the LIKEREC keyword</td></tr>
<tr><td rowspan="2">Definition specification keywords</td><td>ALIAS</td><td>Use the alternate field names for the subfields of the externally-described data structure</td></tr>
<tr><td>RTNPARM</td><td>Specifies that the return value for the procedure should be handled as a hidden parameter</td></tr>
</table>

Table 2. New Language Elements Since V6R1 (continued)

Language Unit	Element	Description
Built-in functions	%PARMNUM	Returns the ordinal number of the parameter in the parameter list
	%SCANRPL	Scans for all occurrences of a value within a string and replaces them with another value
XML-INTO options	datasubf	Name a subfield that will receive the text data for an XML element that also has attributes
	countprefix	Specifies the prefix for the names of the additional subfields that receive the number of RPG array elements or non-array subfields set by the XML-INTO operation

What's New in V6R1

This section describes the enhancements made to ILE RPG in V6R1.

THREAD(*CONCURRENT)

When THREAD(*CONCURRENT) is specified on the Control specification of a module, it provides ability to run concurrently in multiple threads:

- Multiple threads can run in the module at the same time.
- By default, static variables will be defined so that each thread will have its own copy of the static variable.
- Individual variables can be defined to be shared by all threads using STATIC(*ALLTHREAD).
- Individual procedures can be serialized so that only one thread can run them at one time, by specifying SERIALIZE on the Procedure-Begin specification.

Ability to define a main procedure which does not use the RPG cycle

Using the MAIN keyword on the Control specification, a subprocedure can be identified as the program entry procedure. This allows an RPG application to be developed where none of the modules uses the RPG cycle.

Files defined in subprocedures

Files can be defined locally in subprocedures. I/O to local files can only be done with data structures; I and O specifications are not allowed in subprocedures, and the compiler does not generate I and O specifications for externally described files. By default, the storage associated with local files is automatic; the file is closed when the subprocedure returns. The STATIC keyword can be used to indicate that the storage associated with the file is static, so that all invocations of the subprocedure will use the same file, and if the file is open when the subprocedure returns, it will remain open for the next call to the subprocedure.

Qualified record formats

When a file is defined with the QUALIFIED keyword, the record formats must be qualified by the file name, MYFILE.MYFMT. Qualified files do not have I and O specifications generated by the compiler; I/O can only be done through data structures.

Files defined like other files

Using the LIKEFILE keyword, a file can be defined to use the same settings as another File specification, which is important when passing a file as a parameter. If the file is externally-described, the QUALIFIED keyword is implied. I/O to the new file can only be done through data structures.

Files passed as parameters

A prototyped parameter can be defined as a File parameter using the LIKEFILE keyword. Any file related through the same LIKEFILE definition may be passed as a parameter to the procedure. Within the called procedure or program, all supported operations can be done on the file; I/O can only be done through data structures.

EXTDESC keyword and EXTFILE(*EXTDESC)

The EXTDESC keyword identifies the file to be used by the compiler at compile time to obtain the external decription of the file; the filename is specified as a literal in one of the forms 'LIBNAME/FILENAME' or 'FILENAME'. This removes the need to provide a compile-time override for the file.

The EXTFILE keyword is enhanced to allow the special value *EXTDESC, indicating that the file specified by EXTDESC is also to be used at runtime.

EXTNAME to specify the library for the externally-described data structure

The EXTNAME keyword is enhanced to allow a literal to specify the library for the external file. EXTNAME('LIBNAME/FILENAME') or EXTNAME('FILENAME') are supported. This removes the need to provide a compile-time override for the file.

EXFMT allows a result data structure

The EXFMT operation is enhanced to allow a data structure to be specified in the result field. The data structure must be defined with usage type *ALL, either as an externally-described data structure for the record format (EXTNAME(file:fmt:*ALL), or using LIKEREC of the record format (LIKEREC(fmt:*ALL).

Larger limits for data structures, and character, UCS-2 and graphic variables

- Data structures can have a size up to 16,773,104.
- Character definitions can have a length up to 16,773,104. (The limit is 4 less for variable length character definitions.)
- Character definitions can have a length up to 16,773,104. (The limit is 4 less for variable length character definitions.)
- UCS-2 definitions can have a length up to 8,386,552 UCS-2 characters. (The limit is 2 less for variable length UCS-2 definitions.)
- Graphic definitions can have a length up to 8,386,552 DBCS characters. (The limit is 2 less for variable length graphic definitions.)
- The VARYING keyword allows a parameter of either 2 or 4 indicating the number of bytes used to hold the length prefix.

%ADDR(varying : *DATA)

The %ADDR built-in function is enhanced to allow *DATA as the second parameter to obtain the address of the data part of a variable length field.

Larger limit for DIM and OCCURS

An array or multiple-occurrence data structure can have up to 16,773,104 elements, provided that the total size is not greater than 16,773,104.

Larger limits for character, UCS-2 and DBCS literals

- Character literals can now have a length up to 16380 characters.
- UCS-2 literals can now have a length up to 8190 UCS-2 characters.
- Graphic literals can now have a length up to 16379 DBCS characters.

TEMPLATE keyword for files and definitions

The TEMPLATE keyword can be coded for file and variable definitions to indicate that the name will only be used with the LIKEFILE, LIKE, or LIKEDS keyword to define other files or variables. Template definitions are useful when defining types for prototyped calls, since the compiler only uses them at compile time to help define other files and variables, and does not generate any code related to them.

Template data structures can have the INZ keyword coded for the data structure and its subfields, which will ease the use of INZ(*LIKEDS).

Relaxation of some UCS-2 rules

The compiler will perform some implicit conversion between character, UCS-2 and graphic values, making it unnecessary to code %CHAR, %UCS2 or %GRAPH in many cases. This enhancement is also available through PTFs for V5R3 and V5R4. Implicit conversion is now supported for

- Assignment using EVAL and EVALR.
- Comparison operations in expressions.
- Comparison using fixed form operations IFxx, DOUxx, DOWxx, WHxx, CASxx, CABxx, COMP.
- Note that implicit conversion was already supported for the conversion operations MOVE and MOVEL.

UCS-2 variables can now be initialized with character or graphic literals without using the %UCS2 built-in function.

Eliminate unused variables from the compiled object

New values *UNREF and *NOUNREF are added to the OPTION keyword for the CRTBNDRPG and CRTRPGMOD commands, and for the OPTION keyword on the Control specification. The default is *UNREF. *NOUNREF indicates that unreferenced variables should not be generated into the RPG module. This can reduce program size, and if imported variables are not referenced, it can reduce the time taken to bind a module to a program or service program.

PCML can now be stored in the module

Program Call Markup Language (PCML) can now be stored in the module as well as in a stream file. By using combinations of the PGMINFO command parameter and/or the new PGMINFO keyword for the Control specification, the RPG programmer can choose where the PCML information should go. If the PCML information is placed in the module, it can later be retrieved using the QBNRPII API. This enhancement is also available through PTFs for V5R4, but only through the Control specification keyword.

Table 3. Changed Language Elements Since V5R4

Language Unit	Element	Description
Control specification keywords	OPTION(*UNREF | *NOUNREF)	Specifies that unused variables should not be generated into the module.
	THREAD(*CONCURRENT)	New parameter *CONCURRENT allows running concurrently in multiple threads.
File specification keywords	EXTFILE(*EXTDESC)	Specifies that the value of the EXTDESC keyword is also to be used for the EXTFILE keyword.
Built-in functions	%ADDR(varying-field : *DATA)	Can now be used to obtain the address of the data portion of a varying-length variable.
Definition specification keywords	DIM(16773104)	An array can have up to 16773104 elements.
	EXTNAME('LIB/FILE')	Allows a literal for the file name. The literal can include the library for the file.
	OCCURS(16773104)	A multiple-occurrence data structure can have up to 16773104 elements.
	VARYING{(2|4)}	Can now take a parameter indicating the number of bytes for the length prefix.
Definition specifications	Length entry	Can be up to 9999999 for Data Structures, and definitions of type A, C or G. (To define a longer item, the LEN keyword must be used.)
Input specifications	Length entry	Can be up to 99999 for alphanumeric fields, and up to 99998 for UCS-2 and Graphic fields.
Calculation specifications	Length entry	Can be up to 99999 for alphanumeric fields.
Operation codes	EXFMT format { result-ds }	Can have a data structure in the result entry.

Table 4. New Language Elements Since V5R4

Language Unit	Element	Description
Control specification keywords	MAIN(subprocedure-name)	Specifies the program-entry procedure for the program.
	PGMINFO(*NO | *PCML { : *MODULE })	Indicates whether Program Information is to be placed directly in the module.

Table 4. New Language Elements Since V5R4 (continued)

Language Unit	Element	Description
File specification keywords	STATIC	Indicates that a local file retains its program state across calls to a subprocedure.
	QUALIFIED	Indicates that the record format names of the file are qualified by the file name, FILE.FMT.
	LIKEFILE(filename)	Indicates that the file is defined the same as another file.
	TEMPLATE	Indicates that the file is only to be used for later LIKEFILE definitions.
	EXTDESC(constant-filename)	Specifies the external file used at compile time for the external definitions.
Definition specification keywords	STATIC(*ALLTHREAD)	Indicates that the same instance of the static variable is used by all threads running in the module.
	LIKEFILE(filename)	Indicates that the parameter is a file.
	TEMPLATE	Indicates that the definition is only to be used for LIKE or LIKEDS definitions.
	LEN(length)	Specifies the length of a data structure, or a definition of type A, C or G.
Procedure specification keywords	SERIALIZE	Indicates that the procedure can be run by only one thread at a time.

What's New in V5R4?

The following list describes the enhancements made to ILE RPG in V5R4:

New operation code EVAL-CORR

`EVAL-CORR{(EH)}` *`ds1 = ds2`*

New operation code EVAL-CORR assigns data and null-indicators from the subfields of the source data structure to the subfields of the target data structure. The subfields that are assigned are the subfields that have the same name and compatible data type in both data structures.

For example, if data structure DS1 has character subfields A, B, and C, and data structure DS2 has character subfields B, C, and D, statement EVAL-CORR DS1 = DS2; will assign data from subfields DS2.B and DS2.C to DS1.B and DS1.C. Null-capable subfields in the target data structure that are affected by the EVAL-CORR operation will also have their null-indicators assigned from the null-indicators of the source data structure's subfields, or set to *OFF, if the source subfield is not null-capable.

```
//  DS1 subfields        DS2 subfields
//    s1  character        s1 packed
//    s2  character        s2 character
//    s3  numeric
//    s4  date             s4 date
//                         s5 character
EVAL-CORR  ds1 = ds2;
// This EVAL-CORR operation is equivalent to the following EVAL operations
//    EVAL  ds1.s2 = ds2.s2
//    EVAL  ds1.s4 = ds2.s4
// Other subfields either appear in only one data structure (S3 and S5)
// or have incompatible types (S1).
```

EVAL-CORR makes it easier to use result data structures for I/O operations to externally-described files and record formats, allowing the automatic transfer of data between the data structures of different record formats, when the record formats have differences in layout or minor differences in the types of the subfields.

New prototyped parameter option OPTIONS(*NULLIND)

When OPTIONS(*NULLIND) is specified for a parameter, the null-byte map is passed with the parameter, giving the called procedure direct access to the null-byte map of the caller's parameter.

New builtin function %XML

%XML (*xmldocument* { : *options* })

The %XML builtin function describes an XML document and specifies options to control how the document should be parsed. The **xmldocument** parameter can be a character or UCS-2 expression, and the value may be an XML document or the name of an IFS file containing an XML document. If the value of the **xmldocument** parameter has the name of a file, the "doc=file" option must be specified.

New builtin function %HANDLER

%HANDLER (*handlingProcedure* : *communicationArea*)

%HANDLER is used to identify a procedure to handle an event or a series of events. %HANDLER does not return a value, and it can only be specified as the first operand of XML-SAX and XML-INTO.

The first operand, *handlingProcedure*, specifies the prototype of the handling procedure. The return value and parameters specified by the prototype must match the parameters required for the handling procedure; the requirements are determined by the operation that %HANDLER is specified for.

The second operand, *communicationArea,* specifies a variable to be passed as a parameter on every call to the handling procedure. The operand must be an exact match for the first prototyped parameter of the handling procedure, according to the same rules that are used for checking prototyped parameters passed by reference. The communication-area parameter can be any type, including arrays and data structures.

New operation code XML-SAX

XML-SAX{ (e) } %HANDLER(*eventHandler* : *commArea*) %XML(*xmldocument* { : *saxOptions* });

XML-SAX initiates a SAX parse for the XML document specified by the %XML builtin function. The XML-SAX operation begins by calling an XML parser which begins to parse the document. When the parser discovers an event such as finding the start of an element, finding an attribute name, finding the end of an element etc., the parser calls the *eventHandler* with parameters describing the event. The *commArea* operand is a variable that is passed as a parameter to the *eventHandler* providing a way for the XML-SAX operation code to communicate with the handling procedure. When the *eventHandler* returns, the parser continues to parse until it finds the next event and calls the *eventHandler* again.

New operation code XML-INTO

```
XML-INTO{ (EH) } variable   %XML(xmlDoc { : options });
XML-INTO{ (EH) } %HANDLER(handler : commArea ) %XML(xmlDoc { : options });
```

XML-INTO reads the data from an XML document in one of two ways:

- directly into a variable
- gradually into an array parameter that it passes to the procedure specified by %HANDLER.

Various options may be specified to control the operation.

The first operand specifies the target of the parsed data. It can contain a variable name or the % HANDLER built-in function.

The second operand contains the %XML builtin function specifying the source of the XML document and any options to control how the document is parsed. It can contain XML data or it can contain the location of the XML data. The doc option is used to indicate what this operand specifies.

```
// Data structure "copyInfo" has two subfields, "from"
// and "to".  Each of these subfields has two subfields
// "name" and "lib".
// File cpyA.xml contains the following XML document
// <copyinfo>
//    <from><name>MASTFILE</name><lib>CUSTLIB</lib></from>
//    <to><name>MYFILE</name><lib>*LIBL</lib>
// <copyinfo>
xml-into copyInfo %XML('cpyA.xml' : 'doc=file');
// After the XML-INTO operation, the following
// copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo.to    .name = 'MYFILE    ' .lib = '*LIBL     '
```

Use the PREFIX keyword to remove characters from the beginning of field names

```
PREFIX('' : number_of_characters)
```

When an empty character literal (two single quotes specified with no intervening characters) is specified as the first parameter of the PREFIX keyword for File and Definition specifications, the specified number of characters is removed from the field names. For example if a file has fields XRNAME, XRIDNUM, and XRAMOUNT, specifying `PREFIX('':2)`on the File specification will cause the internal field names to be NAME, IDNUM, and AMOUNT.

If you have two files whose subfields have the same names other than a file-specific prefix, you can use this feature to remove the prefix from the names of the subfields of externally-described data structures defined from those files. This would enable you to use EVAL-CORR to assign the same-named subfields from one data structure to the other. For example, if file FILE1 has a field F1NAME and file FILE2 has a field F2NAME, and `PREFIX('':2)` is specified for externally-described data structures DS1 for FILE1 and DS2 for FILE2, then the subfields F1NAME and F2NAME will both become NAME. An EVAL-CORR operation between data structures DS1 and DS2 will assign the NAME subfield.

New values for the DEBUG keyword

```
DEBUG { ( *INPUT  *DUMP  *XMLSAX *NO *YES ) }
```

The DEBUG keyword determines what debugging aids are generated into the module. *NO and *YES are existing values. *INPUT, *DUMP and *XMLSAX provide more granularity than *YES.

***INPUT**
Fields that appear only on input specifications are read into the program fields during input operations.

***DUMP**
DUMP operations without the (A) extender are performed.

***XMLSAX**
An array of SAX event names is generated into the module to be used while debugging a SAX event handler.

***NO**
Indicates that no debugging aids are to be generated into the module. Specifying DEBUG(*NO) is the same as omitting the DEBUG keyword.

***YES**
This value is kept for compatibility purposes. Specifying DEBUG(*YES) is the same as specifying DEBUG without parameters, or DEBUG(*INPUT : *DUMP).

Syntax-checking for free-form calculations

In SEU, free-form statements are now checked for correct syntax.

Improved debugging support for null-capable subfields of a qualified data structure

When debugging qualified data structures with null-capable subfields, the null-indicators are now organized as a similar data structure with an indicator subfield for every null-capable subfield. The name of the data structure is _QRNU_NULL_*data_structure_name*, for example _QRNU_NULL_MYDS. If a subfield of the data structure is itself a data structure with null-capable subfields, the null- indicator data structure will similarly have a data structure subfield with indicator subfields. For example, if data structure DS1 has null-capable subfields DS1.FLD1, DS1.FLD2, and DS1.SUB.FLD3, you can display all the null-indicators in the entire data structure using the debug instruction.

```
===> EVAL _QRNU_NULL_DS
> EVAL _QRNU_NULL_DS1
  _QRNU_NULL_DS1.FLD1 = '1'
  _QRNU_NULL_DS1.FLD2 = '0'
  _QRNU_NULL_DS1.SUB.FLD3 = '1'
===> EVAL _QRNU_NULL_DS.FLD2
  _QRNU_NULL_DS1.FLD2 = '0'
===> EVAL _QRNU_NULL_DS.FLD2 = '1'
===> EVAL DSARR(1).FLD2
  DSARR(1).FLD2 = 'abcde'

===> EVAL _QRNU_NULL_DSARR(1).FLD2

  _QRNU_NULL_DSARR(1).FLD2 = '0'
```

Change to end-of-file behaviour with shared files

If a module performs a keyed sequential input operation to a shared file and it results in an EOF condition, and a different module sets the file cursor using a positioning operation such as SETLL, a subsequent sequential input operation by the first module may be successfully done. Before this change, the first RPG module ignored the fact that the other module had repositioned the shared file.

This change in behaviour is available with PTFs for releases V5R2M0 (SI13932) and V5R3M0 (SI14185).

Table 5. Changed Language Elements Since V5R3

Language Unit	Element	Description
Control specification keywords	DEBUG(*INPUT | *DUMP *XMLSAX | *NO | *YES)	New parameters *INPUT, *DUMP and *XMLSAX give more options for debugging aids.
File specification keywords	`PREFIX('':2)`	An empty literal may be specified as the first parameter of the PREFIX keyword, allowing characters to be removed from the beginning of names.
Definition specification keywords	OPTIONS(*NULLIND)	Indicates that the null indicator is passed with the parameter.
	`PREFIX('':2)`	An empty literal may be specified as the first parameter of the PREFIX keyword, allowing characters to be removed from the beginning of names.

Table 6. New Language Elements Since V5R3

Language Unit	Element	Description
Built-in functions	%HANDLER(prototype: parameter)	Specifies a handling procedure for an event.
	%XML(document{:options})	Specifies an XML document and options to control the way it is parsed.
Operation codes	EVAL-CORR	Assigns data and null-indicators from the subfields of the source data structure to the subfields of the target data structure.
	XML-INTO	Reads the data from an XML document directly into a program variable.
	XML-SAX	Initiates a SAX parse of an XML document.

What's New in V5R3?

The following list describes the enhancements made to ILE RPG in V5R3:

- **New builtin function %SUBARR:**

New builtin function %SUBARR allows assignment to a sub-array or returning a sub-array as a value.

Along with the existing %LOOKUP builtin function, this enhancements enables the implementation of dynamically sized arrays with a varying number of elements.

%SUBARR(array : start) specifies array elements array(start) to the end of the array

%SUBARR(array : start : num) specifies array elements array(start) to array(start + num - 1)

Example:

```
// Copy part of an array to another array:
resultArr = %subarr(array1:start:num);
// Copy part of an array to part of another array:
%subarr(Array1:x:y) = %subarr(Array2:m:n);
// Sort part of an array
sorta %subarr(Array3:x:y);

// Sum part of an array
sum = %xfoot(%subarr(Array4:x:y));
```

- **The SORTA operation code is enhanced to allow sorting of partial arrays.**

  When %SUBARR is specified in factor 2, the sort only affects the partial array indicated by the %SUBARR builtin function.

- **Direct conversion of date/time/timestamp to numeric, using %DEC:**

  %DEC is enhanced to allow the first parameter to be a date, time or timestamp, and the optional second parameter to specify the format of the resulting numeric value.

  Example:

```
D numDdMmYy       s              6p 0
D date            s               d    datfmt(*jul)
 date = D'2003-08-21';
 numDdMmYy = %dec(date : *dmy);   // now numDdMmYy = 210803
```

- **Control specification CCSID(*CHAR : *JOBRUN) for correct conversion of character data at runtime:**

  The Control specification CCSID keyword is enhanced to allow a first parameter of *CHAR. When the first parameter is *CHAR, the second parameter must be *JOBRUN. CCSID(*CHAR : *JOBRUN) controls the way character data is converted to UCS-2 at runtime. When CCSID(*CHAR:*JOBRUN) is specified, character data will be assumed to be in the job CCSID; when CCSID(*CHAR : *JOBRUN) is not specified, character data will be assumed to be in the mixed-byte CCSID related to the job CCSID.

- **Second parameter for %TRIM, %TRIMR and %TRIML indicating what characters to trim:**

  %TRIM is enhanced to allow an optional second parameter giving the list of characters to be trimmed.

  Example:

```
trimchars = '*-.';
data = '***a-b-c-.'
result = %trim(data : trimchars);
// now result = 'a-b-c'.  All * - and . were trimmed from the ends of the data
```

- **New prototype option OPTIONS(*TRIM) to pass a trimmed parameter:**

  When OPTIONS(*TRIM) is specified on a prototyped parameter, the data that is passed be trimmed of leading and trailing blanks. OPTIONS(*TRIM) is valid for character, UCS-2 and graphic parameters defined with CONST or VALUE. It is also valid for pointer parameters defined with OPTIONS(*STRING). With OPTIONS(*STRING : *TRIM), the passed data will be trimmed even if a pointer is passed on the call.

Example:

```
D proc            pr
D   parm1                        5a    const options(*trim)
D   parm2                        5a    const options(*trim : *rightadj)
D   parm3                        5a    const varying options(*trim)
D   parm4                         *    value options(*string : *trim)
D   parm5                         *    value options(*string : *trim)
D ptr             s               *
D data            s             10a
D fld1            s              5a

 /free
   data = ' rst ' + x'00';
   ptr = %addr(data);

   proc (' xyz ' : ' @#$ ' : ' 123 ' : ' abc ' : ptr);
   // the called procedure receives the following parameters
   //    parm1 = 'xyz  '
   //    parm2 = '  @#$'
   //    parm3 = '123'
   //    parm4 = a pointer to 'abc.' (where . is x'00')
   //    parm5 = a pointer to 'rst.' (where . is x'00')
```

- **Support for 63 digit packed and zoned decimal values**

  Packed and zoned data can be defined with up to 63 digits and 63 decimal positions. The previous limit was 31 digits.

- **Relaxation of the rules for using a result data structure for I/O to externally-described files and record formats**
  - The result data structure for I/O to a record format may be an externally-described data structure.
  - A data structure may be specified in the result field for I/O to an externally-described file name for operation codes CHAIN, READ, READE, READP and READPE.

  Examples:

  1. The following program writes to a record format using from an externally-described data structure.

```
Foutfile   o    e           k disk
D outrecDs      e ds                  extname(outfile) prefix(O_)
/free
  O_FLD1 = 'ABCDE';
  O_FLD2 = 7;
  write outrec outrecDs;
  *inlr = *on;
/end-free
```

  2. The following program reads from a multi-format logical file into data structure INPUT which contains two overlapping subfields holding the fields of the respective record formats.

```
Flog       if   e           k disk    infds(infds)
D infds           ds
D   recname             261    270
D input           ds                  qualified
D   rec1                              likerec(rec1) overlay(input)
D   rec2                              likerec(rec2) overlay(input)
 /free
  read log input;
  dow not %eof(log);
   dsply recname;
```

```
    if recname = 'REC1';
    // handle rec1
    elseif recname = 'REC2';
     // handle rec2
    endif;
    read log input;
   enddo;
   *inlr = *on;
  /end-free
```

- If a program/module performs a keyed sequential input operation to a shared file and it results in an EOF condition, a subsequent sequential input operation by the same program/module may be attempted. An input request is sent data base and if a record is available for input, the data is moved into the program/module and the EOF condition is set off.
- **Support for new environment variables for use with RPG programs calling Java methods**
  - **QIBM_RPG_JAVA_PROPERTIES** allows RPG users to explicitly set the java properties used to start the JVM

    This environment variable must be set before any RPG program calls a Java method in a job.

    This environment variable has contains Java options, separated and terminated by some character that does not appear in any of the option strings. Semicolon is usually a good choice.

    Examples:

    1. **Specifying only one option:** If the system's default JDK is 1.3, and you want your RPG programs to use JDK 1.4, set environment variable QIBM_RPG_JAVA_PROPERTIES to

    ```
    '-Djava.version=1.4;'
    ```

    Note that even with just one option, a terminating character is required. This example uses the semicolon.

    2. **Specifying more than one option:** If you also want to set the os400.stdout option to a different value than the default, you could set the environment variable to the following value:

    ```
    '-Djava.version=1.4!-Dos400.stdout=file:mystdout.txt!'
    ```

    This example uses the exclamation mark as the separator/terminator. Note: This support is also available in V5R1 and V5R2 with PTFs. V5R1: SI10069, V5R2: SI10101.
  - **QIBM_RPG_JAVA_EXCP_TRACE** allows RPG users to get the exception trace when an RPG call to a Java method ends with an exception

    This environment variable can be set, changed, or removed at any time.

    If this environment variable contains the value 'Y', then when a Java exception occurs during a Java method call from RPG, or a called Java method throws an exception to its caller, the Java trace for the exception will be printed. By default, it will be printed to the screen, and may not be possible to read. To get it printed to a file, set the Java option os400.stderr. (This would have to be done in a new job; it could be done by setting the QIBM_RPG_JAVA_PROPERTIES environment variable to

    ```
    '-Dos400.stderr=file:stderr.txt;'
    ```
- **An RPG preprocessor enabling the SQL preprocessor to handle conditional compilation and nested /COPY**

  When the RPG compiler is called with a value other than *NONE for parameter PPGENOPT, it will behave as an RPG preprocessor. It will generate a new source file rather than generating a program. The new source file will contain the original source lines that are accepted by the conditional compilation directives such as /DEFINE and /IF. It will also have the source lines from files included by /COPY statements, and optionally it will have the source lines included by /INCLUDE statements. The new source file will have the comments from the original source file if PPGENOPT(*DFT) or PPGENOPT(*NORMVCOMMENT) is specified.When the SQL precompiler is called with a value other than *NONE for new parameter RPGPPOPT, the precompiler will use this RPG preprocessor to handle /COPY, the conditional compilation directives and possibly the /INCLUDE directive. This will allow SQLRPGLE source to have nested /COPY statements, and conditionally used statements.

Table 7. Changed Language Elements Since V5R2

Language Unit	Element	Description
Control specification keywords	CCSID(*GRAPH:parameter | *UCS2:number | *CHAR:*JOBRUN)	Can now take a first parameter of *CHAR, with a second parameter of *JOBRUN, to control how character data is treated at runtime.
Built-in Functions	%DEC(expression {format})	Can now take a parameter of type Date, Time or Timestamp
	%TRIM(expression:expression)	Can now take a second parameter indicating the set of characters to be trimmed
Definition Specification Keywords	OPTIONS(*TRIM)	Indicates that blanks are to be trimmed from passed parameters
Definition Specifications	Length and decimal place entries	The length and number of decimal places can be 63 for packed and zoned fields.
Input specifications	Length entry	The length can be 32 for packed fields and 63 for zoned fields.
	Decimal place entry	The number of decimal places can be 63 for packed and zoned fields.
Calculation specifications	Length and decimal place entries	The length and number of decimal places can be 63 for packed and zoned fields.
	CHAIN, READ, READE, READP, AND READPE operations	Allow a data structure to be specified in the result field when Factor 2 is the name of an externally-described file.
	CHAIN, READ, READC, READE, READP, READPE, WRITE, UPDATE operations	Allow an externally-described data structure to be specified in the result field when Factor 2 is the name of an externally-described record format.
	SORTA operation	Now has an extended Factor 2, allowing %SUBARR to be specified.

Table 8. New Language Elements Since V5R2

Language Unit	Element	Description
Built-in Functions	%SUBARR(array:starting element {:number of elements})	Returns a section of the array, or allows a section of the array to be modified.

What's New in V5R2?

The following list describes the enhancements made to ILE RPG in V5R2:

- Conversion from character to numeric

  Built-in functions %DEC, %DECH, %INT, %INTH, %UNS, %UNSH and %FLOAT are enhanced to allow character parameters. For example, %DEC('-12345.67' : 7 : 2) returns the numeric value `-12345.67`.
- Bitwise logical built-in functions

  %BITAND, %BITOR, %BITXOR and %BITNOT allow direct bit manipulation within RPG expressions.
- Complex data structures

  Data structure definition is enhanced to allow arrays of data structures and subfields of data structures defined with LIKEDS that are themselves data structures. This allows the coding of complex structures such as arrays of arrays, or arrays of structures containing subarrays of structures.

  ```
  Example:   family(f).child(i).hobbyInfo.pets(p).type = 'dog';
             family(f).child(i).hobbyInfo.pets(p).name = 'Spot';
  ```

  In addition, data structures can be defined the same as a record format, using the new LIKEREC keyword.
- Enhanced externally-described data structures

  Externally-described data structures can hold the programmer's choice of input, output, both, key or all fields. Currently, externally-described data structures can only hold input fields.
- Enhancments to keyed I/O

  Programmers can specify search arguments in keyed Input/Output operations in /FREE calculations in two new ways:

  1. By specifying the search arguments (which can be expressions) in a list.
  2. By specifying a data structure which contains the search arguments.

  ```
  Examples: D custkeyDS    e ds        extname(custfile:*key)
             /free
                CHAIN  (keyA : keyB : key3) custrec;
                CHAIN  %KDS(custkeyDS) custrec;
  ```
- Data-structure result for externally-described files

  A data structure can be specified in the result field when using I/O operations for externally-described files. This was available only for program-described files prior to V5R2. Using a data structure can improve performance if there are many fields in the file.
- UPDATE operation to update only selected fields

  A list of fields to be updated can be specified with an UPDATE operation. Tthis could only be done by using exception output prior to V5R2.

  ```
  Example: update record %fields(salary:status).
  ```
- 31 digit support

Supports packed and zoned numeric data with up to 31 digits and decimal places. This is the maximum length supported by DDS. Only 30 digits and decimal places were supported prior to V5R2.

- Performance option for FEOD

  The FEOD operation is enhanced by supporting an extender N which indicates that the operation should simply write out the blocked buffers locally, without forcing a costly write to disk.
- Enhanced data area access

  The DTAARA keyword is enhanced to allow the name and library of the data area to be determined at runtime
- New assignment operators

  The new assignment operators +=, -=, *=, /=, **= allow a variable to be modified based on its old value in a more concise manner.

  ```
  Example: totals(current_customer) += count;
  ```

  This statement adds "count" to the value currently in "totals(current_customer)" without having to code "totals(current_customer)" twice.
- IFS source files

  The ILE RPG compiler can compile both main source files and /COPY files from the IFS. The /COPY and /INCLUDE directives are enhanced to support IFS file names.
- Program Call Markup Language (PCML) generation

  The ILE RPG compiler will generate an IFS file containing the PCML, representing the parameters to the program (CRTBNDRPG) or to the exported procedures (CRTRPGMOD).

Table 9. Changed Language Elements Since V5R1

Language Unit	Element	Description
Built-in functions	%DEC(expression)	Can now take parameters of type character.
	%DECH(expression)	
	%FLOAT(expression)	
	%INT(expression)	
	%INTH(expression)	
	%UNS(expression)	
	%UNSH(expression)	
Definition specification keywords	DTAARA({*VAR:}data-area-name)	The data area name can be a name, a character literal specifying 'LIBRARY/NAME' or a character variable which will determine the actual data area at runtime.
	DIM	Allowed for data structure specifications.
	LIKEDS	Allowed for subfield specifications.
	EXTNAME(filename{:extrecname} {:*ALL|*INPUT|*OUTPUT|*KEY})	The optional "type" parameter controls which type of field is extracted for the externally-described data structure.
Definition Specifications	Length and decimal place entries	The length and number of decimal places can be 31 for packed and zoned fields.

Table 9. Changed Language Elements Since V5R1 (continued)

Language Unit	Element	Description
Operation codes	CHAIN, DELETEREADE, READPE, SETGT, SETLL	In free-form operations, Factor 1 can be a list of key values.
	CHAIN, READ, READC, READE, READP, READPE, UPDATE, WRITE	When used with externally-described files or record formats, a data structure may be specified in the result field.
	UPDATE	In free-form calculations, the final argument can contain a list of the fields to be updated.
	FEOD	Operation extender N is allowed. This indicates that the unwritten buffers must be made available to the database, but not necessarily be written to disk.
Calculation specifications	Length and decimal place entries	The length and number of decimal places can be 31 for packed and zoned fields.

Table 10. New Language Elements Since V5R1

Language Unit	Element	Description
Expressions	Assignment Operators += -= *= /= **=	When these assignment operators are used, the target of the operation is also the first operand of the operation.
Control Specification Keywords	DECPREC(30|31)	Controls the precision of decimal intermediate values for presentation, for example, for %EDITC and %EDITW
Definition specification keywords	LIKEREC(intrecname{:*ALL|*INPUT|*OUTPUT|*KEY})	Defines a data structure whose subfields are the same as a record format.
Built-in functions	%BITAND(expression : expression)	Returns a result whose bits are on if the corresponding bits of the operands are both on.
	%BITNOT(expression)	Returns a result whose bits are the inverse of the bits in the argument.
	%BITOR(expression : expression)	Returns a result whose bits are on if either of the corresponding bits of the operands is on.
	%BITXOR(expression : expression)	Returns a result whose bits are on if exactly one of the corresponding bits of the operands is on.
	%FIELDS(name{:name...})	Used in free-form "UPDATE to specify the fields to be updated.
	%KDS(data structure)	Used in free-form keyed operation codes CHAIN, SETLL, SETGT, READE and READPE, to indicate that the keys for the operation are in the data structure.

What's New in V5R1?

The ILE RPG compiler is part of the IBM IBM Rational Development Studio for System i product, which now includes the C/C++ and COBOL compilers, and the Application Development ToolSet tools.

The major enhancements to RPG IV since V4R4 are easier interfacing with Java, new built-in functions, free form calculation specifications, control of which file is opened, qualified subfield names, and enhanced error handling.

The following list describes these enhancements:

- Improved support for calls between Java and ILE RPG using the Java Native Interface (JNI):
  - A new data type: Object
  - A new definition specification keyword: CLASS
  - The LIKE definition specification keyword has been extended to support objects.
  - The EXTPROC definition specification keyword has been extended to support Java procedures.
  - New status codes.
- New built-in functions:
  - Functions for converting a number into a duration that can be used in arithmetic expressions: %MSECONDS, %SECONDS, %MINUTES, %HOURS, %DAYS, %MONTHS, and %YEARS.
  - The %DIFF function, for subtracting one date, time, or timestamp value from another.
  - Functions for converting a character string (or date or timestamp) into a date, time, or timestamp: %DATE, %TIME, and %TIMESTAMP.
  - The %SUBDT function, for extracting a subset of a date, time, or timestamp.
  - Functions for allocating or reallocating storage: %ALLOC and %REALLOC.
  - Functions for finding an element in an array: %LOOKUP, %LOOKUPGT, %LOOKUPGE, %LOOKUPLT, and %LOOKUPLE.
  - Functions for finding an element in a table: %TLOOKUP, %TLOOKUPGT, %TLOOKUPGE, %TLOOKUPLT, and %TLOOKUPLE.
  - Functions for verifying that a string contains only specified characters (or finding the first or last exception to this rule): %CHECK and %CHECKR
  - The %XLATE function, for translating a string based on a list of from-characters and to-characters.
  - The %OCCUR function, for getting or setting the current occurrence in a multiple-occurrence data structure.
  - The %SHTDN function, for determining if the operator has requested shutdown.
  - The %SQRT function, for calculating the square root of a number.
- A new free-form syntax for calculation specifications. A block of free-form calculation specifcations is delimited by the compiler directives /FREE and /END-FREE
- You can specify the EXTFILE and EXTMBR keywords on the file specification to control which external file is used when a file is opened.
- Support for qualified names in data structures:
  - A new definition specification keyword: QUALIFIED. This keyword specifies that subfield names will be qualified with the data structure name.
  - A new definition specification keyword: LIKEDS. This keyword specifies that subfields are replicated from another data structure. The subfield names will be qualified with the new data structure name. LIKEDS is allowed for prototyped parameters; it allows the parameter's subfields to be used directly in the called procedure.
  - The INZ definition specification keyword has been extended to allow a data structure to be initialized based on its parent data structure.
- Enhanced error handling:

– Three new operation codes (MONITOR, ON-ERROR, and ENDMON) allow you to define a group of operations with conditional error handling based on the status code.

Other enhancements have been made to this release as well. These include:
- You can specify parentheses on a procedure call that has no parameters.
- You can specify that a procedure uses ILE C or ILE CL calling conventions, on the EXTPROC definition specification keyword.
- The following /DEFINE names are predefined: *VnRnMn, *ILERPG, *CRTBNDRPG, and *CRTRPGMOD.
- The search string in a %SCAN operation can now be longer than string being searched. (The string will not be found, but this will no longer generate an error condition.)
- The parameter to the DIM, OCCURS, and PERRCD keywords no longer needs to be previously defined.
- The %PADDR built-in function can now take either a prototype name or an entry point name as its argument.
- A new operation code, ELSEIF, combines the ELSE and IF operation codes without requiring an additional ENDIF.
- The DUMP operation code now supports the A extender, which means that a dump is always produced - even if DEBUG(*NO) was specified.
- A new directive, /INCLUDE, is equivalent to /COPY except that /INCLUDE is not expanded by the SQL preprocessor. Included files cannot contain embedded SQL or host variables.
- The OFLIND file-specification keyword can now take any indicator, including a named indicator, as an argument.
- The LICOPT (licensed internal code options) keyword is now available on the CRTRPGMOD and CRTBNDRPG commands.
- The PREFIX file description keyword can now take an uppercase character literal as an argument. The literal can end in a period, which allows the file to be used with qualified subfields.
- The PREFIX definition specification keyword can also take an uppercase character literal as an argument. This literal cannot end in a period.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 11. Changed Language Elements Since V4R4

Language Unit	Element	Description
Built-in functions	%CHAR(expression{:format})	The optional second parameter specifies the desired format for a date, time, or timestamp. The result uses the format and separators of the specified format, not the format and separators of the input.
	%PADDR(prototype-name)	This function can now take either a prototype name or an entry point name as its argument.

Table 11. Changed Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Definition specification keywords	EXTPROC(*JAVA:class-name:proc-name)	Specifies that a Java method is called.
	EXTPROC(*CL:proc-name)	Specifies a procedure that uses ILE CL conventions for return values.
	EXTPROC(*CWIDEN:proc-name)	Specifies a procedure that uses ILE C conventions with parameter widening.
	EXTPROC(*CNOWIDEN:proc-name)	Specifies a procedure that uses ILE C conventions without parameter widening.
	INZ(*LIKEDS)	Specifies that a data structure defined with the LIKEDS keyword inherits the initialization from its parent data structure.
	LIKE(object-name)	Specifies that an object has the same class as another object.
	PREFIX(character-literal{:number})	Prefixes the subfields with the specified character literal, optionally replacing the specified number of characters.
File specification keywords	OFLIND(name)	This keyword can now take any named indicator as a parameter.
	PREFIX(character-literal{:number})	Prefixes the subfields with the specified character literal, optionally replacing the specified number of characters.
Operation codes	DUMP (A)	This operation code can now take the A extender, which causes a dump to be produced even if DEBUG(*NO) was specified.

Table 12. New Language Elements Since V4R4

Language Unit	Element	Description
Data types	Object	Used for Java objects
Compiler directives	/FREE ... /END-FREE	The /FREE... /END-FREE compiler directives denote a free-form calculation specifications block.
	/INCLUDE	Equivalent to /COPY, except that it is not expanded by the SQL preprocessor. Can be used to inlcude nested files that are within the copied file. The copied file cannot have embedded SQlL or host variables.
Definition specification keywords	CLASS(*JAVA:class-name)	Specifies the class for an object.
	LIKEDS(dsname)	Specifies that a data structure, prototyped parameter, or return value inherits the subfields of another data strucutre.
	QUALIFIED	Specifies that the subfield names in a data structure are qualified with the data structure name.
File specification keywords	EXTFILE(filename)	Specifies which file is opened. The value can be a literal or a variable. The default file name is the name specified in position 7 of the file specification. The default library is *LIBL.
	EXTMBR(membername)	Specifies which member is opened. The value can be a literal or a variable. The default is *FIRST.

Table 12. New Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Built-in functions	%ALLOC(num)	Allocates the specified amount of storage.
	%CHECK(comparator:base{:start})	Finds the first character in the base string that is not in the comparator.
	%CHECKR(comparator:base{:start})	Finds the last character in the base string that is not in the comparator.
	%DATE(expression{:date-format})	Converts the expression to a date.
	%DAYS(num)	Converts the number to a duration, in days.
	%DIFF(op1:op2:unit)	Calculates the difference (duration) between two date, time, or timestamp values in the specified units.
	%HOURS(num)	Converts the number to a duration, in hours.
	%LOOKUPxx(arg:array{:startindex {:numelems}})	Finds the specified argument, or the specified type of near-match, in the specified array.
	%MINUTES(num)	Converts the number to a duration, in minutes.
	%MONTHS(num)	Converts the number to a duration, in months.
	%MSECONDS(num)	Converts the number to a duration, in microseconds.
	%OCCUR(dsn-name)	Sets or gets the current position of a multiple-occurrence data structure.
	%REALLOC(pointer:number)	Reallocates the specified amount of storage for the specified pointer.
	%SECONDS(num)	Converts the number to a duration, in seconds.
	%SHTDN	Checks if the system operator has requested shutdown.
	%SQRT(numeric-expression)	Calculates the square root of the specified number.
	%SUBDT(value:unit)	Extracts the specified portion of a date, time, or timestamp value.
	%THIS	Returns an Object value that contains a reference to the class instance on whose behalf the native method is being called.
	%TIME(expression{:time-format})	Converts the expression to a time.
	%TIMESTAMP(expression {:*ISO | *ISO0})	Converts the expression to a timestamp.
	%TLOOKUP(arg:search-table {:alt-table})	Finds the specified argument, or the specified type of near-match, in the specified table.
	%XLATE(from:to:string{:startpos})	Translates the specified string, based on the from-string and to-string.
	%YEARS(num)	Converts the number to a duration, in years.

Table 12. New Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Operation codes	MONITOR	Begins a group of operations with conditional error handling.
	ON-ERROR	Performs conditional error handling, based on the status code.
	ENDMON	Ends a group of operations with conditional error handling.
	ELSEIF	Equivalent to an ELSE operation code followed by an IF operation code.
CRTBNDRPG and CRTRPGMOD keywords	LICOPT(options)	Specifies Licensed Internal Code options.

What's New in V4R4?

The major enhancements to RPG IV since V4R2 are the support for running ILE RPG modules safely in a threaded environment, the new 3-digit and 20-digit signed and unsigned integer data types, and support for a new Universal Character Set Version 2 (UCS-2) data type and for conversion between UCS-2 fields and graphic or single-byte character fields.

The following list describes these enhancements:

- Support for calling ILE RPG procedures from a threaded application, such as Domino® or Java™.
  - The new control specification keyword THREAD(*SERIALIZE) identifies modules that are enabled to run in a multithreaded environment. Access to procedures in the module is serialized.
- Support for new 1-byte and 8-byte integer data types: 3I and 20I signed integer, and 3U and 20U unsigned integer
  - These new integer data types provide you with a greater range of integer values and can also improve performance of integer computations, taking full advantage of the 64-bit AS/400 RISC processor.
  - The new 3U type allows you to more easily communicate with ILE C procedures that have single-byte character (char) return types and parameters passed by value.
  - The new INTPREC control specification keyword allows you to specify 20-digit precision for intermediate values of integer and unsigned binary arithmetic operations in expressions.
  - Built-in functions %DIV and %REM have been added to support integer division and remainder operations.
- Support for new Universal Character Set Version 2 (UCS-2) or Unicode data type
  - The UCS-2 (Unicode) character set can encode the characters for many written languages. The field is a character field whose characters are two bytes long.
  - By adding support for Unicode, a single application can now be developed for a multinational corporation, minimizing the necessity to perform code page conversion. The use of Unicode permits the processing of characters in multiple scripts without loss of integrity.
  - Support for conversions between UCS-2 fields and graphic or single-byte character fields using the MOVE and MOVEL operations, and the new %UCS2 and %GRAPH built-in functions.

  – Support for conversions between UCS-2 fields or graphic fields with different Coded Character Set Identifiers (CCSIDs) using the EVAL, MOVE, and MOVEL operations, and the new %UCS2 built-in function.

Other enhancements have been made to this release as well. These include:

- New parameters for the OPTION control specification keyword and on the create commands:
  - *SRCSTMT allows you to assign statement numbers for debugging from the source IDs and SEU sequence numbers in the compiler listing. (The statement number is used to identify errors in the compiler listing by the debugger, and to identify the statement where a run-time error occurs.) *NOSRCSTMT specifies that statement numbers are associated with the Line Numbers of the listing and the numbers are assigned sequentially.
  - Now you can choose not to generate breakpoints for input and output specifications in the debug view with *NODEBUGIO. If this option is selected, a STEP on a READ statement in the debugger will step to the next calculation, rather than stepping through the input specifications.
- New special words for the INZ definition specification keyword:
  - INZ(*EXTDFT) allows you to use the default values in the DDS for initializing externally described data structure subfields.
  - Character variables initialized by INZ(*USER) are initialized to the name of the current user profile.
- The new %XFOOT built-in function sums all elements of a specified array expression.
- The new EVALR operation code evaluates expressions and assigns the result to a fixed-length character or graphic result. The assignment right-adjusts the data within the result.
- The new FOR operation code performs an iterative loop and allows free-form expressions for the initial, increment, and limit values.
- The new LEAVESR operation code can be used to exit from any point within a subroutine.
- The new *NEXT parameter on the OVERLAY(name:*NEXT) keyword indicates that a subfield overlays another subfield at the next available position.
- The new *START and *END values for the SETLL operation code position to the beginning or end of the file.
- The ability to use hexadecimal literals with integer and unsigned integer fields in initialization and free-form operations, such as EVAL, IF, etc.
- New control specification keyword OPENOPT{(*NOINZOFL | *INZOFL)} to indicate whether the overflow indicators should be reset to *OFF when a file is opened.
- Ability to tolerate pointers in teraspace — a memory model that allows more than 16 megabytes of contiguous storage in one allocation.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 13. Changed Language Elements Since V4R2

Language Unit	Element	Description
Control specification keywords	OPTION(*{NO}SRCSTMT)	*SRCSTMT allows you to request that the compiler use SEU sequence numbers and source IDs when generating statement numbers for debugging. Otherwise, statement numbers are associated with the Line Numbers of the listing and the numbers are assigned sequentially.
	OPTION(*{NO}DEBUGIO)	*{NO}DEBUGIO, determines if breakpoints are generated for input and output specifications.
Definition specification keywords	INZ(*EXTDFT)	All externally described data structure subfields can now be initialized to the default values specified in the DDS.
	INZ(*USER)	Any character field or subfield can be initialized to the name of the current user profile.
	OVERLAY(name:*NEXT)	The special value *NEXT indicates that the subfield is to be positioned at the next available position within the overlayed field.
	OPTIONS(*NOPASS *OMIT *VARSIZE *STRING *RIGHTADJ)	The new OPTIONS(*RIGHTADJ) specified on a value or constant parameter in a function prototype indicates that the character, graphic, or UCS-2 value passed as a parameter is to be right adjusted before being passed on the procedure call.
Definition specification positions 33-39 (To Position/Length)	3 and 20 digits allowed for I and U data types	Added to the list of allowed values for internal data types to support 1-byte and 8-byte integer and unsigned data.
Internal data type	C (UCS-2 fixed or variable-length format)	Added to the list of allowed internal data types on the definition specifications. The UCS-2 (Unicode) character set can encode the characters for many written languages. The field is a character field whose characters are two bytes long.
Data format	C (UCS-2 fixed or variable-length format)	UCS-2 format added to the list of allowed data formats on the input and output specifications for program described files.
Command parameter	OPTION	*NOSRCSTMT, *SRCSTMT, *NODEBUGIO, and *DEBUGIO have been added to the OPTION parameter on the CRTBNDRPG and CRTRPGMOD commands.

Table 14. New Language Elements Since V4R2

<table>
<tr><th>Language Unit</th><th>Element</th><th>Description</th></tr>
<tr><td rowspan="5">Control specification keywords</td><td>CCSID(*GRAPH: *IGNORE | *SRC | number)</td><td>Sets the default graphic CCSID for the module. This setting is used for literals, compile-time data and program-described input and output fields and definitions. The default is *IGNORE.</td></tr>
<tr><td>CCSID(*UCS2: number)</td><td>Sets the default UCS-2 CCSID for the module. This setting is used for literals, compile-time data and program-described input and output fields and definitions. The default is 13488.</td></tr>
<tr><td>INTPREC(10 | 20)</td><td>Specifies the decimal precision of integer and unsigned intermediate values in binary arithmetic operations in expressions. The default, INTPREC(10), indicates that 10-digit precision is to be used.</td></tr>
<tr><td>OPENOPT{(*NOINZOFL | *INZOFL)}</td><td>Indicates whether the overflow indicators should be reset to *OFF when a file is opened.</td></tr>
<tr><td>THREAD(*SERIALIZE)</td><td>Indicates that the module is enabled to run in a multithreaded environment. Access to the procedures in the module is to be serialized.</td></tr>
<tr><td>Definition specification keywords</td><td>CCSID(number | *DFT)</td><td>Sets the graphic and UCS-2 CCSID for the definition.</td></tr>
<tr><td rowspan="5">Built-in functions</td><td>%DIV(n:m)</td><td>Performs integer division on the two operands n and m; the result is the integer portion of n/m. The operands must be numeric values with zero decimal positions.</td></tr>
<tr><td>%GRAPH(char-expr | graph-expr | UCS2-expr {: ccsid})</td><td>Converts to graphic data from single-byte character, graphic, or UCS-2 data.</td></tr>
<tr><td>%REM(n:m)</td><td>Performs the integer remainder operation on two operands n and m; the result is the remainder of n/m. The operands must be numeric values with zero decimal positions.</td></tr>
<tr><td>%UCS2(char-expr | graph-expr | UCS2-expr {: ccsid})</td><td>Converts to UCS-2 data from single-byte character, graphic, or UCS-2 data.</td></tr>
<tr><td>%XFOOT(array-expr)</td><td>Produces the sum of all the elements in the specified numeric array expression.</td></tr>
</table>

Table 14. New Language Elements Since V4R2 (continued)

Language Unit	Element	Description
Operation codes	EVALR	Evaluates an assignment statement of the form result=expression. The result will be right-justified.
	FOR	Begins a group of operations and indicates the number of times the group is to be processed. The initial, increment, and limit values can be free-form expressions.
	ENDFOR	ENDFOR ends a group of operations started by a FOR operation.
	LEAVESR	Used to exit from anywhere within a subroutine.

What's New in V4R2?

The major enhancements to RPG IV since V3R7 are the support for variable-length fields, several enhancements relating to indicators, and the ability to specify compile options on the control specifications. These further improve the RPG product for integration with the OS/400 operating system and ILE interlanguage communication.

The following list describes these enhancements:

- Support for variable-length fields

  This enhancement provides full support for variable-length character and graphic fields. Using variable-length fields can simplify many string handling tasks.
- Ability to use your own data structure for INDARA indicators

  Users can now access logical data areas and associate an indicator data structure with each WORKSTN and PRINTER file that uses INDARA, instead of using the *IN array for communicating values to data management.
- Ability to use built-in functions instead of result indicators

  Built-in functions %EOF, %EQUAL, %FOUND, and %OPEN have been added to query the results of input/output operations. Built-in functions %ERROR and %STATUS, and the operation code extender 'E' have been added for error handling.
- Compile options on the control specification

  Compile options, specified through the CRTBNDRPG and CRTRPGMOD commands, can now be specified through the control specification keywords. These compile options will be used on every compile of the program.

In addition, the following new function has been added:

- Support for import and export of procedures and variables with mixed case names
- Ability to dynamically set the DECEDIT value at runtime
- Built-in functions %CHAR and %REPLACE have been added to make string manipulation easier
- New support for externally defined *CMDY, *CDMY, and *LONGJUL date data formats
- An extended range for century date formats

- Ability to define indicator variables
- Ability to specify the current data structure name as the parameter for the OVERLAY keyword
- New status code 115 has been added to indicate variable-length field errors
- Support for application profiling
- Ability to handle packed-decimal data that is not valid when it is retrieved from files using FIXNBR(*INPUTPACKED)
- Ability to specify the BNDDIR command parameter on the CRTRPGMOD command.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 15. Changed Language Elements Since V3R7

Language Unit	Element	Description
Control specification keywords	DECEDIT(*JOBRUN | 'value')	The decimal edit value can now be determined dynamically at runtime from the job or system value.
Definition specification keywords	DTAARA {(data_area_name)}	Users can now access logical data areas.
	EXPORT {(external_name)}	The external name of the variable being exported can now be specified as a parameter for this keyword.
	IMPORT {(external_name)}	The external name of the variable being imported can now be specified as a parameter for this keyword.
	OVERLAY(name{:pos})	The name parameter can now be the name of the current data structure.

Extended century format	*CYMD (cyy/mm/dd)	The valid values for the century character 'c' are now:
		`'c'  Years`
		`-----------------------`
		`0  1900-1999`
		`1  2000-2099`
		`.  .`
		`.  .`
		`.  .`
		`9  2800-2899`

Internal data type	N (Indicator format)	Added to the list of allowed internal data types on the definition specifications. Defines character data in the indicator format.
Data format	N (Indicator format)	Indicator format added to the list of allowed data formats on the input and output specifications for program described files.
Data Attribute	*VAR	Added to the list of allowed data attributes on the input and output specifications for program described files. It is used to specify variable-length fields.

Table 15. Changed Language Elements Since V3R7 (continued)

Language Unit	Element	Description
Command parameter	FIXNBR	The *INPUTPACKED parameter has been added to handle packed-decimal data that is not valid.

Table 16. New Language Elements Since V3R7

Language Unit	New	Description
Control specification keywords	ACTGRP(*NEW | *CALLER | 'activation- group-name')	The ACTGRP keyword allows you to specify the activation group the program is associated with when it is called.
	ALWNULL(*NO | *INPUTONLY | *USRCTL)	The ALWNULL keyword specifies how you will use records containing null-capable fields from externally described database files.
	AUT(*LIBRCRTAUT | *ALL | *CHANGE | *USE | *EXCLUDE | 'authorization-list-name')	The AUT keyword specifies the authority given to users who do not have specific authority to the object, who are not on the authorization list, and whose user group has no specific authority to the object.
	BNDDIR('binding -directory-name' {:'binding-directory-name'...})	The BNDDIR keyword specifies the list of binding directories that are used in symbol resolution.
	CVTOPT(*{NO}DATETIME *{NO}GRAPHIC *{NO}VARCHAR *{NO}VARGRAPHIC)	The CVTOPT keyword is used to determine how the ILE RPG compiler handles date, time, timestamp, graphic data types, and variable-length data types that are retrieved from externally described database files.
	DFTACTGRP(*YES | *NO)	The DFTACTGRP keyword specifies the activation group in which the created program will run when it is called.
	ENBPFRCOL(*PEP | *ENTRYEXIT | *FULL)	The ENBPFRCOL keyword specifies whether performance collection is enabled.
	FIXNBR(*{NO}ZONED *{NO}INPUTPACKED)	The FIXNBR keyword specifies whether decimal data that is not valid is fixed by the compiler.
	GENLVL(number)	The GENLVL keyword controls the creation of the object.
	INDENT(*NONE | 'character-value')	The INDENT keyword specifies whether structured operations should be indented in the source listing for enhanced readability.
	LANGID(*JOBRUN | *JOB | 'language-identifier')	The LANGID keyword indicates which language identifier is to be used when the sort sequence is *LANGIDUNQ or *LANGIDSHR.

Table 16. New Language Elements Since V3R7 (continued)

Language Unit	New	Description
	OPTIMIZE(*NONE | *BASIC | *FULL)	The OPTIMIZE keyword specifies the level of optimization, if any, of the object.
	OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP)	The OPTION keyword specifies the options to use when the source member is compiled.
	PRFDTA(*NOCOL | *COL)	The PRFDTA keyword specifies whether the collection of profiling data is enabled.
	SRTSEQ(*HEX | *JOB | *JOBRUN | *LANGIDUNQ | *LANGIDSHR | 'sort-table-name')	The SRTSEQ keyword specifies the sort sequence table that is to be used in the ILE RPG source program.
	TEXT(*SRCMBRTXT | *BLANK | 'description')	The TEXT keyword allows you to enter text that briefly describes the object and its function.
	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.
	USRPRF(*USER | *OWNER)	The USRPRF keyword specifies the user profile that will run the created program object.
File Description Specification keywords	INDDS(data_structure_name)	The INDDS keyword lets you associate a data structure name with the INDARA indicators for a workstation or printer file.
Definition specification keywords	VARYING	Defines variable-length fields when specified on character data or graphic data.
Built-in functions	%CHAR(graphic, date, time or timestamp expression)	Returns the value in a character data type.
	%EOF{file name}	Returns '1' if the most recent file input operation or write to a subfile (for a particular file, if specified) ended in an end-of-file or beginning-of-file condition; otherwise, it returns '0'.
	%EQUAL{file name}	Returns '1' if the most recent SETLL (for a particular file, if specified) or LOOKUP operation found an exact match; otherwise, it returns '0'.
	%ERROR	Returns '1' if the most recent operation code with extender 'E' specified resulted in an error; otherwise, it returns '0'.

Table 16. New Language Elements Since V3R7 (continued)

Language Unit	New	Description
	%FOUND{file name}	Returns '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); otherwise, it returns '0'.
	%OPEN(file name)	Returns '1' if the specified file is open and '0' if the specified file is closed.
	%REPLACE(replacement string: source string {:start position {:source length to replace}})	Returns the string produced by inserting a **replacement string** into a **source string**, starting at the **start position** and replacing the specified number of characters.
	%STATUS{file name}	If no program or file error occurred since the most recent operation code with extender 'E' specified, it returns 0. If an error occurred, it returns the most recent value set for any program or file status. If a file is specified, the value returned is the most recent status for that file.
Operation code Extender	E	Allows for error handling using the %ERROR and %STATUS built-in functions on the CALLP operation and all operations that allow error indicators.
New century formats	*CMDY (cmm/dd/yy)	To be used by the MOVE, MOVEL, and TEST operations.
	*CDMY (cdd/mm/yy)	To be used by the MOVE, MOVEL, and TEST operations.
New 4-digit year format	*LONGJUL (yyyy/ddd)	To be used by the MOVE, MOVEL, and TEST operations.
Command parameters	PRFDTA	The PRFDTA parameter specifies whether the collection of profiling data is enabled.
	BNDDIR	The BNDDIR parameter was previously only allowed on the CRTBNDRPG command and not on the CRTRPGMOD command, now it is allowed on both commands.

What's New in V3R7?

The major enhancements to RPG IV since V3R6 are the new support for database null fields, and the ability to better control the precision of intermediate results in expressions. Other enhancements include the addition of a floating point data type and support for null-terminated strings. These further improve the RPG product for integration with the OS/400 operating system and ILE interlanguage communication. This means greater flexibility for developing applications.

The following is a list of these enhancements including a number of new built-in functions and usability enhancements:

- Support for database null fields

  This enhancement allows users to process database files which contain null-capable fields, by allowing these fields to be tested for null and set to null.
- Expression intermediate result precision

  A new control specification keyword and new operation code extenders on free-form expression specifications allow the user better control over the precision of intermediate results.
- New floating point data type

  The new floating point data type has a much larger range of values than other data types. The addition of this data type will improve integration with the database and improve interlanguage communication in an ILE environment, specifically with the C and C++ languages.
- Support for null terminated strings

  The new support for null terminated strings improves interlanguage communication. It allows users full control over null terminated data by allowing users to define and process null terminated strings, and to conveniently pass character data as parameters to procedures which expect null terminated strings.
- Pointer addition and subtraction

  Free-form expressions have been enhanced to allow adding an offset to a pointer, subtracting an offset from a pointer, and determining the difference between two pointers.
- Support for long names

  Names longer than 10 characters have been added to the RPG language. Anything defined on the definition or procedure specifications can have a long name and these names can be used anywhere where they fit within the bounds of an entry. In addition, names referenced on any free-form specification may be continued over multiple lines.
- New built-in functions

  A number of new built-in functions have been added to the language which improve the following language facilities:
  - editing (%EDITW, %EDITC, %EDITFLT)
  - scanning strings (%SCAN)
  - type conversions (%INT, %FLOAT, %DEC, %UNS)
  - type conversions with half-adjust (%INTH, %DECH, %UNSH)
  - precision of intermediate results for decimal expressions (%DEC)
  - length and decimals of variables and expressions (%LEN, %DECPOS)
  - absolute value (%ABS)
  - set and test null-capable fields (%NULLIND)
  - handle null terminated strings (%STR)
- Conditional compilation

  RPG IV has been extended to support conditional compilation. This support will include the following:
  - defining conditions (/DEFINE, /UNDEFINE),
  - testing conditions (/IF, /ELSEIF, /ELSE, /ENDIF)
  - stop reading current source file (/EOF)

  – a new command option (DEFINE) to define up to 32 conditions on the CRTBNDRPG and CRTRPGMOD commands.

- Date enhancements

  Several enhancements have been made to improve date handling operations. The TIME operation code is extended to support Date, Time or Timestamp fields in the result field. Moving dates or times from and to character fields no longer requires separator characters. Moving UDATE and *DATE fields no longer requires a format code to be specified. Date fields can be initialized to the system (*SYS) or job (*JOB) date on the definition specifications.

- Character comparisons with alternate collating sequence

  Specific character variables can be defined so that the alternate collating sequence is not used in comparisons.

- Nested /COPY members

  You can now nest /COPY directives. That is, a /COPY member may contain one (or more) /COPY directives which can contain further /COPY directives and so on.

- Storage management

  You can now use the new storage management operation codes to allocate, reallocate and deallocate storage dynamically.

- Status codes for storage management and float underflow errors.

  Two status codes 425 and 426 have been added to indicate storage management errors. Status code 104 was added to indicate that an intermediate float result is too small.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 17. Changed Language Elements Since V3R6

Language Unit	Element	Description
Definition specification keywords	ALIGN	ALIGN can now be used to align float subfields along with the previously supported integer and unsigned alignment.
	OPTIONS(*NOPASS *OMIT *VARSIZE *STRING)	The *STRING option allows you to pass a character value as a null-terminated string.
Record address type	F (Float format)	Added to the list of allowed record address types on the file description specifications. Signals float processing for a program described file.
Internal data type	F (Float format)	Added to the list of allowed internal data types on the definition specifications. Defines a floating point standalone field, parameter, or data structure subfield.
Data format	F (Float format)	Added to the list of allowed data formats on the input and output specifications for program described files.

Table 18. New Language Elements Since V3R6

Language Unit	New	Description
Control specification keywords	COPYNEST('1-2048')	Specifies the maximum depth for nesting of /COPY directives.
	EXPROPTS(*MAXDIGITS | *RESDECPOS)	Expression options for type of precision (default or "Result Decimal Position" precision rules)
	FLTDIV{(*NO | *YES)}	Indicates that all divide operations in expressions are computed in floating point.
Definition specification keywords	ALTSEQ(*NONE)	Forces the normal collating sequence to be used for character comparison even when an alternate collating sequence is specified.
Built-in functions	%ABS	Returns the absolute value of the numeric expression specified as the parameter.
	%DEC & %DECH	Converts the value of the numeric expression to decimal (packed) format with the number of digits and decimal positions specified as parameters. %DECH is the same as %DEC, but with a half adjust applied.
	%DECPOS	Returns the number of decimal positions of the numeric variable or expression. The value returned is a constant, and may be used where a constant is expected.
	%EDITC	This function returns a character result representing the numeric value edited according to the edit code.
	%EDITFLT	Converts the value of the numeric expression to the character external display representation of float.
	%EDITW	This function returns a character result representing the numeric value edited according to the edit word.
	%FLOAT	Converts the value of the numeric expression to float format.
	%INT & %INTH	Converts the value of the numeric expression to integer. Any decimal digits are truncated with %INT and rounded with %INTH.
	%LEN	Returns the number of digits or characters of the variable expression.
	%NULLIND	Used to query or set the null indicator for null-capable fields.
	%SCAN	Returns the first position of the search argument in the source string, or 0 if it was not found.

Table 18. New Language Elements Since V3R6 (continued)

Language Unit	New	Description
	%STR	Used to create or use null-terminated strings, which are very commonly used in C and C++ applications.
	%UNS & %UNSH	Converts the value of the numeric expression to unsigned format. Any decimal digits are truncated with %UNS and rounded with %UNSH.
Operation code Extenders	N	Sets pointer to *NULL after successful DEALLOC
	M	Default precision rules
	R	No intermediate value will have fewer decimal positions than the result ("Result Decimal Position" precision rules)
Operation codes	ALLOC	Used to allocate storage dynamically.
	DEALLOC	Used to deallocate storage dynamically.
	REALLOC	Used to reallocate storage dynamically.

What's New in V3R6/V3R2?

The major enhancement to RPG IV since V3R1 is the ability to code a module with more than one procedure. What does this mean? In a nutshell, it means that you can code an module with one or more prototyped procedures, where the procedures can have return values and run without the use of the RPG cycle.

Writing a module with multiple procedures enhances the kind of applications you can create. Any application consists of a series of logical units that are conceived to accomplish a particular task. In order to develop applications with the greatest flexibility, it is important that each logical unit be as independent as possible. Independent units are:

- Easier to write from the point of view of doing a specific task.
- Less likely to change any data objects other than the ones it is designed to change.
- Easier to debug because the logic and data items are more localized.
- Maintained more readily since it is easier to isolate the part of the application that needs changing.

The main benefit of coding a module with multiple procedures is greater control and better efficiency in coding a modular application. This benefit is realized in several ways. You can now:

- Call procedures and programs by using the same call operation and syntax.
- Define a prototype to provide a check at compile time of the call interface.
- Pass parameters by value or by reference.
- Define a procedure that will return a value and call the procedure within an expression.
- Limit access to data items by defining local definitions of variables.
- Code a module that does not make use of the cycle.
- Call a procedure recursively.

The run-time behavior of the main procedure in a module is the same as that of a V3R1 procedure. The run-time behavior of any subsequent procedures differs somewhat from a V3R1 program, most notably in the areas of procedure end and exception handling. These differences arise because there is no cycle code that is generated for these procedures.

Other enhancements have been made to for this release as well. These include:

- Support for two new integer data types: signed integer (I), and unsigned integer (U)

  The use of the integer data types provides you with a greater range of values than the binary data type. Integer data types can also improve performance of integer computations.
- *CYMD support for the MOVE, MOVEL, and TEST operations

  You can now use the *CYMD date format in certain operations to work with system values that are already in this data format.
- Ability to copyright your programs and modules by using the COPYRIGHT keyword on the control specification

  The copyright information that is specified using this keyword becomes part of the DSPMOD, DSPPGM, or DSPSRVPGM information.
- User control of record blocking using keyword BLOCK

  You can request record blocking of DISK or SEQ files to be done even when SETLL, SETGT, or CHAIN operations are used on the file. You can also request that blocking not be done. Use of blocking in these cases may significantly improve runtime performance.
- Improved PREFIX capability

  Changes to the PREFIX keyword for either file-description and definition specifications allow you to replace characters in the existing field name with the prefix string.
- Status codes for trigger program errors

  Two status codes 1223 and 1224 have been added to indicate trigger program errors.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 19. Changed Language Elements Since V3R1

Language Unit	Element	Description
File description specification keywords	PREFIX(prefix_string {:nbr_of_char_ replaced})	Allows prefixing of string to a field name or a partial rename of the field name
Definition specification keywords	CONST{(constant)}	Specifies the value of a named constant, or indicates that a prototyped parameter that is passed by reference has a constant value
	PREFIX(prefix_string {:nbr_of_char_ replaced})	Allows prefixing of string to a field name or a partial rename of the field name
Operation codes	RETURN	Returns control to the caller, and returns a value, if specified

Table 20. New Language Elements Since V3R1

<table>
<tr><th>Language Unit</th><th>New</th><th>Description</th></tr>
<tr><td>Control specification keywords</td><td>COPYRIGHT('copyright string')</td><td>Allows you to associate copyright information with modules and programs</td></tr>
<tr><td></td><td>EXTBININT{(*NO | *YES)}</td><td>Specifies that binary fields in externally-described files be assigned an integer format during program processing</td></tr>
<tr><td></td><td>NOMAIN</td><td>Indicates that the module has only subprocedures</td></tr>
<tr><td>File description specification keywords</td><td>BLOCK(*YES |*NO)</td><td>Allows you to control whether record blocking occurs (assuming other conditions are met)</td></tr>
<tr><td>Definition specification keywords</td><td>ALIGN</td><td>Specifies whether integer or unsigned fields should be aligned</td></tr>
<tr><td></td><td>EXTPGM(name)</td><td>Indicates the external name of the prototyped program</td></tr>
<tr><td></td><td>EXTPROC(name)</td><td>Indicates the external name of the prototyped procedure</td></tr>
<tr><td></td><td>OPDESC</td><td>Indicates whether operational descriptors are to be passed for the prototyped bound call</td></tr>
<tr><td></td><td>OPTIONS(*NOPASS *OMIT *VARSIZE)</td><td>Specifies various options for prototyped parameters</td></tr>
<tr><td></td><td>STATIC</td><td>Specifies that the local variable is to use static storage</td></tr>
<tr><td></td><td>VALUE</td><td>Specifies that the prototyped parameter is to be passed by value</td></tr>
<tr><td>Built-in functions</td><td>%PARMS</td><td>Returns the number of parameters passed on a call</td></tr>
<tr><td>Operation codes</td><td>CALLP</td><td>Calls a prototyped program or procedure</td></tr>
<tr><td>Specification type</td><td>Procedure specification</td><td>Signals the beginning and end of a subprocedure definition</td></tr>
<tr><td>Definition type</td><td>PR</td><td>Signals the beginning of a prototype definition</td></tr>
<tr><td></td><td>PI</td><td>Signals the beginning of a procedure interface definition</td></tr>
<tr><td></td><td>blank in positions 24-25</td><td>Defines a prototyped parameter</td></tr>
</table>

Part 1. RPG IV Concepts

This section describes some of the basics of RPG IV:

- Symbolic names
- Compiler directives
- RPG IV program cycle
- Indicators
- Error Handling
- Subprocedures
- General file considerations

Chapter 1. Symbolic Names and Reserved Words

The valid character set for the RPG IV language consists of:

- The 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
- RPG IV accepts lowercase letters in symbolic names but translates them to uppercase during compilation
- The numbers 0 1 2 3 4 5 6 7 8 9
- The characters + − * , . ' & / $ # : @ _ > < = ( ) %
- The blank character

Note: The $, #, and @ may appear as different symbols on some codepages. For more information, see the iSeries Information Center globalization topic.

Symbolic Names

A symbolic name is a name that uniquely identifies a specific entity in a program or procedure. In the RPG IV language, symbolic names are used for the following:

- Arrays (see “Array Names” on page 4)
- Conditional compile names (see “Conditional Compile Names” on page 4)
- Data structures (see “Data Structure Names” on page 4)
- Exception output records (see “EXCEPT Names” on page 4)
- Fields (see “Field Names” on page 4)
- Key field lists (see “KLIST Names” on page 4)
- Labels (see “Labels” on page 4)
- Named constants (see “Named Constants” on page 133)
- Parameter lists (see “PLIST Names” on page 5)
- Prototype names (see “Prototype Names” on page 5)
- Record names (see “Record Names” on page 5)
- Subroutines (see “Subroutine Names” on page 5)
- Tables (see “Table Names” on page 5).

The following rules apply to all symbolic names except for deviations noted in the description of each symbolic name:

- The first character of the name must be alphabetic. This includes the characters $, #, and @.
- The remaining characters must be alphabetic or numeric. This includes 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 an RPG IV 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. For more information, see Chapter 11, “About Specifications,” on page 245.

- A symbolic name must be unique within the procedure in which it is defined.

Array Names

The following additional rule applies to array names:

- 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 Compile Names

The symbolic names used for conditional compilation have no relationship to other symbolic names. For example, if you define a file called MYFILE, you may later use /DEFINE to define condition name MYFILE, and you may also use /UNDEFINE to remove condition name MYFILE. This has no effect on the file name MYFILE.

Conditional compile names can be up to 50 characters long.

Data Structure Names

A data structure is an area in storage and is considered to be a character field.

EXCEPT Names

An EXCEPT name is a symbolic name assigned to an exception output record. The following additional rule applies to EXCEPT names:

- The same EXCEPT name can be assigned to more than one output record.

Field Names

The following additional rules apply to field names:

- 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.

KLIST Names

A KLIST name is a symbolic name assigned to a list of key fields.

Labels

A label is a symbolic name that identifies a specific location in a program (for example, the name assigned to a TAG or ENDSR operation).

Named Constants

A named constant is a symbolic name assigned to a constant.

PLIST Names

A PLIST name is a symbolic name assigned to a list of parameters.

Prototype Names

A prototype name is a symbolic name assigned to a prototype definition. This name must be used when calling a prototyped procedure or program. A prototype maybe explicitly specified, or it may be implicitly generated by the compiler from the procedure interface when the procedure is defined in the same module as the call.

Record Names

A record name is a symbolic name assigned to a record format in an externally described file. The following additional rules apply to record names in an RPG IV program:

- If the file is qualified, due to the QUALIFIED or LIKEFILE keyword on the File specification, the record name is specified as a qualified name in the form FILENAME.FMTNAME. The record name must be unique within the other record names of the file.
- If the file is not qualified, the record name is specified without qualification in the form FMTNAME. If the file is a global file, the record name must be unique within the other global names. If the file is a local file in a subprocedure, the record name must be unique within the other local names.

Note: See "RENAME(Ext_format:Int_format)" on page 308 for information on how to handle the situation where the record name conflicts with other names in your RPG program.

Subroutine Names

The name is defined in factor 1 of the BEGSR (begin subroutine) operation.

Table Names

The following additional rules apply to table names:

- A table name can contain from 3 to 10 characters.
- A table name must begin with the letters TAB.
- A table cannot be defined in a subprocedure.

RPG IV Words with Special Functions/Reserved Words

The RPG IV reserved words listed below have special functions within a program.

- The following reserved words allow you to access the job date, or a portion of it, to be used in the program:

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

- The following reserved words can be used for numbering the pages of a report, for record sequence numbering, or to sequentially number output fields:

  PAGE

  PAGE1-PAGE7
- Figurative constants are implied literals that allow specifications without referring to length:

  *BLANK/*BLANKS

  *ZERO/*ZEROS

  *HIVAL

  *LOVAL

  *NULL

  *ON

  *OFF

  *ALLX'x1..'

  *ALLG'oK1K2i'

  *ALL'X..'
- The following reserved words are used for positioning database files. *START positions to beginning of file and *END positions to end of file.

  *END

  *START
- The following reserved words allow RPG IV indicators to be referred to as data:

  *IN

  *INxx
- The following are special words used with date and time:

  *CDMY

  *CMDY

  *CYMD

  *DMY

  *EUR

  *HMS

  *ISO

  *JIS

  *JOB

  *JOBRUN

  *JUL

  *LONGJUL

  *MDY

  *SYS

  *USA

  *YMD
- The following are special words used with translation:

  *ALTSEQ

  *EQUATE

  *FILE

  *FTRANS

- *PLACE allows repetitive placement of fields in an output record. (See "*PLACE" on page 409 for more information.)
- *ALL allows all fields that are defined for an externally described file to be written on output. (See "Rules for Figurative Constants" on page 135 for more information on *ALL)
- The following are special words used within expressions:

  AND

  NOT

  OR

  Note: NOT can only be used within expressions. It cannot be used as a name anywhere in the source.
- The following are special words used with parameter passing:

  *NOPASS

  *OMIT

  *RIGHTADJ

  *STRING

  *TRIM

  *VARSIZE
- The following special words aid in interpreting the event parameter in an event handling procedure for the XML-SAX operation code:

  XML_ATTR_UCS2_REF

  XML_ATTR_NAME

  XML_ATTR_PREDEF_REF

  XML_ATTR_CHARS

  XML_CHARS

  XML_COMMENT

  XML_UCS2_REF

  XML_PREDEF_REF

  XML_DOCTYPE_DECL

  XML_ENCODING_DECL

  XML_END_CDATA

  XML_END_DOCUMENT

  XML_END_ELEMENT

  XML_END_PREFIX_MAPPING

  XML_EXCEPTION

  XML_PI_TARGET

  XML_PI_DATA

  XML_STANDALONE_DECL

  XML_START_CDATA

  XML_START_DOCUMENT

  XML_START_ELEMENT

  XML_START_PREFIX_MAPPING

  XML_UNKNOWN_ATTR_REF

  XML_UNKNOWN_REF

  XML_VERSION_INFO

XML_END_ATTR

User Date Special Words

The user date special words (UDATE, *DATE, UMONTH, *MONTH, UDAY, *DAY, UYEAR, *YEAR) allow the programmer to supply a date for the program at run time. The user date special words access the job date that is specified in the job description. The user dates can be written out at output time; UDATE and *DATE can be written out using the Y edit code in the format specified by the control specification.

(For a description of the job date, see the*Work Management* manual.)

Rules for User Date

Remember the following rules when using the user date:

- UDATE, when specified in positions 30 through 43 of the output specifications, prints a 6-character numeric date field. *DATE, when similarly specified, prints an 8-character (4-digit year portion) numeric date field. These special words can be used in three different date formats:

  Month/day/year

  Year/month/day

  Day/month/year

  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)

  Note that the DATEDIT keyword also controls the format of the Y edit code.

  If this keyword is not specified, the default is *MDY.
- For an interactive job or batch program, the user date special words are set to the value of the job date when the program starts running in the system. The value of the user date special words are not updated during program processing, even if the program runs past midnight or if the job date is changed. Use the TIME operation code to obtain the time and date while the program is running.
- UMONTH, *MONTH, UDAY, *DAY, and UYEAR when specified in positions 30 through 43 of the output specifications, print a 2-position numeric date field. *YEAR can be used to print a 4-position numeric date field. Use UMONTH or *MONTH to print the month only, UDAY or *DAY to print the day only, and UYEAR or *YEAR to print the year only.
- UDATE and *DATE can be edited when they are written if the Y edit code is specified in position 44 of the output specifications. The "DATEDIT(fmt{separator})" on page 262 keyword on the control specification determines the format and the separator character to be inserted; for example, 12/31/88, 31.12.88., 12/31/1988.
- UMONTH, *MONTH, UDAY, *DAY, UYEAR and *YEAR cannot be edited by the Y edit code in position 44 of the output specifications.
- The user date fields cannot be modified. This means they cannot be used:
  - In the result field of calculations

  - As factor 1 of PARM operations
  - As the factor 2 index of LOOKUP operations
  - With blank after in output specifications
  - As input 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.
- User date fields are not date data type fields but are numeric fields.

PAGE, PAGE1-PAGE7

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 needed for numbering different types of output pages or for numbering 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.

Rules for PAGE, PAGE1-PAGE7

Remember the following rules when using the PAGE fields:

- When a PAGE field is specified in the output specifications, without being defined elsewhere, it is assumed to be a four-digit, numeric field with zero decimal positions.
- Page numbering, unless otherwise specified, starts with 0001; and 1 is automatically added for each new page.
- To start at a page number other than 1, set the value of the PAGE field to one less than the starting page number. For example, if numbering starts with 24, enter a 23 in the PAGE field. The PAGE field can be of any length but must have zero decimal positions (see Figure 1 on page 10).
- Page numbering can be restarted at any point in a job. The following methods can be used to reset the PAGE field:
  - 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 specifications (see Figure 2 on page 10). When the output indicator is on, the PAGE field will be reset to 1. Output indicators cannot be used to control the printing of a PAGE field, because a PAGE field is always written.
  - Specify the PAGE field as an input field as shown in Figure 1 on page 10.
- Leading zeros are automatically suppressed (Z edit code is assumed) when a PAGE field is printed unless an edit code, edit word, or data format (P/B/L/R in position 52) has been specified. Editing and the data format override the suppression of leading zeros. When the PAGE field is defined in input and calculation specifications, it is treated as a field name in the output specifications and zero suppression is not automatic.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IINPUT     PG  50   1  CP
I                                  2    5 0PAGE
```

Figure 1. Page Record Description

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
OFilename++DF..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     H    L1                      01
O               15      PAGE          1     75
```

Figure 2. Resetting the PAGE Fields to Zero

Chapter 2. Compiler Directives

The compiler directive statements /FREE... /END-FREE denote a free-form calculation specification block. The compiler directive statements /TITLE, /EJECT, /SPACE, /COPY, 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 compilation directive statements /DEFINE, /UNDEFINE, /IF, /ELSEIF, /ELSE, /ENDIF, and /EOF allow you to select or omit source records. The compiler directive statements must precede any compile-time array or table records, translation records, and alternate collating sequence records (that is, ** 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 399 for information on using free-form statements.

/TITLE (Positions 7-12)

Use the compiler directive /TITLE to specify heading information (such as security classification or titles) that is to appear at the top of each page of the compiler listing. The following entries are used for /TITLE:

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

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 RPG specification encountered 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.

The /TITLE statement causes a skip to the next page before the title is printed. The /TITLE statement is not printed on the compiler listing.

/EJECT (Positions 7-12)

Positions	**Entry**
7-12	/EJECT

13-49	Blank
50-100	Comments

Enter /EJECT in positions 7 through 12 to indicate that subsequent specifications are to begin on a new page of the compiler listing. Positions 13 through 49 of the /EJECT statement must be blank. The remaining positions may be used for comments. If the spool file is already at the top of a new page, /EJECT will not advance to a new page. /EJECT is not printed on the compiler listing.

/SPACE (Positions 7-12)

Use the compiler directive /SPACE to control line spacing within the source section of the compiler listing. The following entries are used for /SPACE:

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 on the compiler listing. The number must be left-adjusted.
17-49	Blank
50-100	Comments

If the number specified in positions 14 through 16 is greater 112, 112 will be used as the /SPACE value. If the number specified in positions 14 through 16 is greater than the number of lines remaining on the current page, subsequent specifications begin at the top of the next page.

/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.

/COPY or /INCLUDE

The /COPY and /INCLUDE directives have the same purpose and the same syntax, but are handled differently by the SQL precompiler. If your program does not have embedded SQL, you can freely choose which directive to use. If your program has embedded SQL, see "Using /COPY, /INCLUDE in Source Files with Embedded SQL" on page 14 for information about 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, with the file being compiled. The inserted files may contain any valid specification including /COPY and /INCLUDE up to the maximum nesting depth specified by the COPYNEST keyword (32 when not specified).

/COPY and /INCLUDE files can be either physical files or IFS files. To specify a physical file, code your /COPY and /INCLUDE statement in the following way :
- /COPY or /INCLUDE followed by exactly one space followed by the file name or path
- when specifying a physical file, the library, file, and member name, can be in one of these formats:

```
libraryname/filename,membername
filename,membername
membername
```

– A member name must be specified.
– If a file name is not specified, QRPGLESRC is assumed.
– 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.

- When specifying an IFS (Integrated File System) file, the path can be either absolute (beginning with /) or relative.
  – The path can be enclosed in single or double quotes. If the path contains blanks, it must be enclosed in quotes.
  – If the path does not end with a suffix (for example ".txt"), the compiler will search for the file as named, and also for files with suffixes of ".rpgle" or ".rpgleinc".
  – See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for information on using IFS /COPY files.
- Optionally, at least one space and a comment.

/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.

TIP

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.

Figure 3 shows some examples of the /COPY and /INCLUDE directive statements.

```
C/COPY MBR1  1

I/INCLUDE SRCFIL,MBR2  2

O/COPY SRCLIB/SRCFIL,MBR3  3

O/INCLUDE "SRCLIB!"/"SRC>3","MBR¬3"  4

O/COPY /dir1/dir2/file.rpg  5

O/COPY /dir1/dir2/file  6

O/COPY dir1/dir2/file.rpg  7

O/COPY "ifs file containing blanks"  8

O/COPY 'ifs file containing blanks'  8
```

Figure 3. Examples of the /COPY and /INCLUDE Compiler Directive Statements

1 Copies from member MBR1 in source file QRPGLESRC. The current library list is used to search for file QRPGLESRC. If the file is not found in the library list, the search will proceed to the IFS, looking for file MBR1, MBR1.rpgle or MBR1.rpgleinc in the include search path. See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for information on using IFS source files.

2 Copies from member MBR2 in file SRCFIL. The current library list is used to search for file SRCFIL. Note that the comma is used to separate the file name from the member name. If the file is not found in the library list, the search will proceed to the IFS, looking for file *SRCFIL, MBR*1 in the include search path, possibly with the .rpgle or .rpgleinc suffixes.

3 Copies from member MBR3 in file SRCFIL in library SRCLIB or from the IFS file *SRCFIL, MBR3* in directory SRCLIB.

4 Copies from member "MBR¬3" in file "SRC>3" in library "SRCLIB!"

5 Copies from the IFS file file.rpg in directory /dir1/dir2.

6 Copies from file, or file.rpgleinc or file.rpgle in directory /dir1/dir2

7 Copies from the IFS file file.rpg in directory dir1/dir2, searching for directory dir1/dir2 using the IFS search path.

8 Copies from a file whose name contains blanks.

Results of the /COPY or /INCLUDE during Compile

During compilation, the specified file members are merged into the program at the point where the /COPY or /INCLUDE statement occurs. All members will appear in the COPY member table.

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.

TIP

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.

For an example of how to prevent multiple inclusion, see Figure 4 on page 19.

Using /COPY, /INCLUDE in Source Files with Embedded SQL

The /COPY and /INCLUDE directives are identical except that they are handled differently by the SQL precompiler.

The way the /COPY and /INCLUDE directives are handled by the SQL precompiler is different depending on the RPG preprocessor options parameter (RPGPPOPT) specified on the CRTSQLRPGI command. Refer to "Coding SQL statements in ILE RPG applications" in the Embedded SQL Programming topic or the CRTSQLRPGI command in the CL topic for more information.

Conditional Compilation Directives

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

- Condition-names can be added or removed from a list of currently defined conditions using the defining condition directives /DEFINE and /UNDEFINE.
- Condition expressions DEFINED(condition-name) and NOT DEFINED(condition-name) are used within testing condition /IF groups.
- Testing condition 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.

Note: The command parameter DEFINE can be used to predefine up to 32 conditions on the CRTBNDRPG and CRTRPGMOD commands.

/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(condtion-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 Environment

***ILERPG**
This condition is defined if your program is being compiled by the ILE RPG IV compiler (the compiler described in this document).

```
 * This module is to be defined on different platforms.  With
 * the ILE RPG compiler, the BNDDIR keyword is used to
 * indicate where procedures can be found.  With a different
 * compiler, the BNDDIR keyword might not be valid.
 /IF DEFINED(*ILERPG)
H BNDDIR('QC2LE')
 /ENDIF
```

To learn what conditions are available with another version of the RPG IV compiler, consult the reference for the compiler. For example, for VisualAge RPG see *VisualAge RPG Language Reference*, SC09-2451-06.

Conditions Relating to the Command Being Used

***CRTBNDRPG**
This condition is defined if your program is being compiled by the CRTBNDRPG command, which creates a program.

```
 /IF DEFINED(*CRTBNDRPG)
H DFTACTGRP(*NO)
 /ENDIF
```

***CRTRPGMOD**
This condition is defined if your program is being compiled by the CRTRPGMOD command, which creates a module.

```
 * This code might appear in a generic Control specification
 * contained in a /COPY file.  The module that contains the
 * main procedure would define condition THIS_IS_MAIN before
 * coding the /COPY directive.

 * If the CRTRPGMOD command is not being used, or if
 * THIS_IS_MAIN is defined, the NOMAIN keyword will not
 * be used in this Control specification.

 /IF DEFINED(*CRTRPGMOD)
 /IF NOT DEFINED(THIS_IS_MAIN)
H NOMAIN
 /ENDIF
 /ENDIF
```

Conditions Relating to the Target Release

***VxRxMx**
This condition is defined if your program is being compiled for a version that is greater than or equal to the release in the condition, starting with *V4R4M0 (Version 4 Release 4 Modification 0).

Use this condition if you will run the same program on different target releases, and want to take advantage of features that are not available in every release. Support for this condition is available starting with *V4R4M0 systems with the appropriate PTF installed.

```
 /IF DEFINED(*V5R1M0)

 * Specify code that is valid in V5R1M0 and subsequent releases

I/INCLUDE SRCFIL,MBR2

 /ELSE
 * Specify code that is available in V4R4M0

I/COPY SRCFIL,MBR2

 /ENDIF
```

Condition Expressions

A condition 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. Otherwse, 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 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.

TIP

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).

The following is an example of the /EOF directive.

```
*-----------------------------------------------------------------
* Main source file
*-----------------------------------------------------------------
....
/IF DEFINED(READ_XYZ)                        1
/COPY XYZ
/ENDIF                                       2
....
*-----------------------------------------------------------------
* /COPY file XYZ
*-----------------------------------------------------------------
/IF DEFINED(XYZ_COPIED)                      3
/EOF
/ELSE
/DEFINE XYZ_COPIED
 D .....
/ENDIF
```

Figure 4. /EOF Directive

The first time this /COPY member is read, XYZ_COPIED will not be defined, so the /EOF will not be considered.

The second time this member is read, XYZ_COPIED is defined, so the /EOF is processed. The /IF DEFINED(XYZ_COPIED) (3) is considered ended, and the file is closed. However, the /IF DEFINED(READ_XYZ) (1) from the main source member is still active until its own /ENDIF (2) is reached.

Handling of Directives by the RPG Preprocessor

The handling of compiler directives by the RPG preprocessor depends on the options specified on the PPGENOPT parameter on the compile command. There are several actions the preprocessor can take on a particular directive:

- The directive may be kept in the generated source file (indicated by "keep" in the table below)
- The directive may be removed from the generated source file (indicated by "remove" in the table below)
- The directive may be kept in the generated source file, but as a comment (indicated by "comment" in the table below)

In general, with option *RMVCOMMENT, only the directives neccessary for successful compilation are output to the generated source file. With option NORMVCOMMENT, the directives not necessary for successful compilation of the generated source file are converted into comments.

The following table summarizes how each directive is handled by the preprocessor for the various PPGENOPT parameter values:

	*RMVCOMMENT		*NORMVCOMMENT	
Directive	*EXPINCLUDE	*NOEXPINCLUDE	*EXPINCLUDE	*NOEXPINCLUDE
/COPY	remove	remove	comment	comment
/DEFINE	remove	keep	comment	keep
/EJECT	remove	remove	keep	keep
/ELSE	remove	remove	comment	comment
/ELSEIF	remove	remove	comment	comment
/END-EXEC	keep	keep	keep	keep
/END-FREE	keep	keep	keep	keep
/ENDIF	remove	remove	comment	comment
/EOF	remove	remove	comment	comment
/EXEC	keep	keep	keep	keep
/FREE	keep	keep	keep	keep
/IF	remove	remove	comment	comment
/INCLUDE	remove	keep	comment	keep
/SPACE	remove	remove	keep	keep
/TITLE	remove	remove	keep	keep
/UNDEFINE	remove	keep	comment	keep

Chapter 3. Procedures and the Program Logic Cycle

A procedure is a collection of statements that can be called and run.

There are three kinds of procedures in RPG: regular subprocedures, linear-main procedures and cycle-main procedures. RPG source programs can be compiled into one of three kinds of modules depending on the types of procedures present, and as indicated by the presence of the NOMAIN or MAIN keyword on the Control specification: Cycle, Nomain, or Linear-main modules.

The term "subprocedure" is used to denote both regular subprocedures and linear-main procedures.

An RPG source program can be divided into these sections which contain procedures:

- **Main source section:** The source lines from the first line in the source program up to the first Procedure specification. In a cycle module, this section may contain calculation specifications (standard or free-form) which make up a cycle-main procedure. A cycle-main procedure is implied even if there are no calculation specifications in this section. This kind of procedure does not have Procedure-Begin and Procedure-End specifications to identify it.

  A cycle module may be designed without sub-procedures, and thus have no separate Procedure section.
- **Procedure section:** Zero or one linear-main procedures, and one or more regular sub-procedures, defined within the source program. Each procedure begins with a Procedure-Begin specification, and ends with a Procedure-End specification.

  The linear-main procedure is indicated by the use of the MAIN keyword on a Control specification, making it a special kind of sub-procedure.

Subprocedure Definition

A **subprocedure** is a procedure defined after the main source section.

A subprocedure differs from a cycle-main procedure in several respects, the main difference being that a subprocedure does not (and cannot) use the RPG cycle while running.

A subprocedure may have a corresponding prototype in the definition specifications of the main source section. If specified, the prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters. If not specified, the prototype is implicitly generated from the procedure interface.

TIP

Although it is optional to specify a prototype within the module that defines the procedure, it should not be considered optional when it is exported from the module, and the procedure will be called from other RPG modules. In this case, a prototype should be specified in a copy file and copied into the module that defines the subprocedure and into every module that calls the subprocedure.

Figure 5 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
 /free
    Result = Term1 ** 2 * 17
                 + Term2 * 7                               5
                 + Term3;
    return Result * 45 + 23;
 /end-free
P                 E                                        6
```

Figure 5. Example of a Subprocedure

1 A Prototype which specifies the name, return value if any, and parameters if any. Since the procedure is not exported from this module, it is optional to specify the prototype.

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. If the prototype had not been specified, the procedure-interface definition would be used by the compiler to implicitly define the prototype.

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.

No cycle code is generated for subprocedures. Consequently, you cannot code:
- Prerun-time and compile-time arrays and tables
- *DTAARA definitions
- Total calculations

The calculation specifications are processed only once and the procedure returns at the end of the calculation specifications. See “Subprocedure Calculations” on page 43 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.

Procedure Interface Definition

If a prototyped procedure has call parameters or a return value, then it must have a procedure interface definition. If a prototype has been specified for the procedure, the **procedure interface definition** is a repeat of the prototype information within the definition of a procedure. Otherwise, the procedure interface definition is used to implicitly define the prototype for the procedure. The procedure interface definition 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 157.

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 to the subprocedure. 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 cycle-main procedure.
- 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.

```
 * Main procedure definitions
D Fld1           S             1A
D Fld2           S             1A

 * Define a global key field list with 2 fields, Fld1 and Fld2
C     global_kl     KLIST
C                   KFLD                    Fld1
C                   KFLD                    Fld2

 * Subprocedure Section
P Subproc        B
D Fld2           S             1A

 * local_kl has one global kfld (fld1) and one local (fld2)
C     local_kl      KLIST
C                   KFLD                    Fld1
C                   KFLD                    Fld2

 * Even though Fld2 is defined locally in the subprocedure,
 * the global Fld2 is used by the global_kl, since global KLISTs
 * always use global fields.  As a result, the assignment to the
 * local Fld2 will NOT affect the CHAIN operation.

C                   EVAL      Fld1 = 'A'
C                   EVAL      Fld2 = 'B'
C     global_kl     SETLL     file

 * Local KLISTs use global fields only when there is no local
 * field of that name.  local_kl uses the local Fld2 and so the
 * 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
```

Figure 6. Scope of Key Fields Inside a Module

For more information on scope, see "Scope of Definitions" on page 126.

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.

If you do not require the improvements offered by subprocedures, you may want to use a subroutine because an EXSR operation is usually faster than a call to a subprocedure.

Program Flow in RPG Modules: Cycle Versus Linear

The ILE RPG compiler supplies part of the logic for an RPG module. Depending on the type of module you choose, this supplied logic will control a large or small part of the control flow of your module. By default, an RPG module will include the full RPG Cycle, which begins with the *INIT phase and ends with the *TERM phase. The other two types of RPG modules do not include the full RPG Cycle; the only remnant of the RPG cycle is the module initialization, which is similar to the *INIT phase. The ILE RPG compiler supplies additional implicit logic that is separate from the RPG cycle; for example, the implicit opening and closing of local files in subprocedures.

All ILE RPG modules can have one or more procedures.

The three types of RPG modules are distinguished by the nature of the main procedure in the module.

A program or a service program can consist of multiple modules, each of which can have an RPG main procedure. If an RPG module is selected to be the program-entry module of a program, then you call the main procedure using a program call. If an RPG module is not the program-entry module of a program, or if it is a module in a service program, then you call its main procedure using a bound call. Calling a main procedure through a bound call is only available for cycle-main procedures; if a module contains a linear-main procedure and that module is not selected to be a program-entry module, than that procedure cannot be called.

A module with a cycle-main procedure
The module contains a cycle-main procedure and zero or more subprocedures. The cycle-main procedure includes the logic for the full RPG cycle. A cycle-main procedure can be called through a bound call, or through a program call. See “Cycle Module” on page 27 and “Program Cycle” on page 31 for more information.

A module with a linear-main procedure
The module contains a linear-main procedure and zero or more ordinary subprocedures. The linear-main procedure is identified by the MAIN keyword on the Control specification. The main procedure itself is coded as a subprocedure (with Procedure specifications). The linear-main procedure can only be called through a program call; it cannot be called using a bound call.

Note: Other than the way it is called, the linear-main procedure is considered to be a subprocedure.

The module does not include the logic for the RPG cycle. See “Linear Main Module” on page 30 for more information.

A module with no main procedure
The NOMAIN keyword on the Control specification indicates that there is

no main procedure in the module. The module contains only subprocedures. The module does not include the logic for the RPG cycle.

This type of module cannot be the program-entry module of a program, since it has no main procedure.

See "NOMAIN Module" on page 30 for more information.

Table 21. Summary of RPG module types

Module Type	Keyword	Cycle Features Allowed	Main Procedure	Initialization of global variables, opening of files, and UDS data areas	Implicit closing of global files and unlocking of data areas
Cycle-main		Yes	Implicitly defined in the main source section	• When the first procedure in the module is called after the activation group is created. • When the main procedure is called, if the main procedure previously ended with LR on, or ended abnormally.	When the main procedure ends with LR on, or ends abnormally.
Linear-main	MAIN	No	Explicitly defined with the MAIN keyword and Procedure specifications	When the main procedure is first called after the activation group is created, or if somehow a sub-procedure is called first.	Never
No main	NOMAIN	No	None, indicated by the presence of the NOMAIN keyword	When the first procedure in the module is called after the activation group is created	Never

Cycle Module

A cycle module has a cycle-main procedure which uses the RPG Program Cycle; the procedure is implicitly specified in the main source section . (See "Program Cycle" on page 31.) You do not need to code anything special to define the main procedure; it consists of everything before the first Procedure specification. The parameters for the cycle-main procedure can be coded using a procedure interface and an optional prototype in the global Definition specifications, or using a *ENTRY PLIST in the cycle-main procedure's calculations.

The name of the cycle-main procedure must be the same as the name of the module being created. You can either use this name for the prototype and procedure interface, or specify this name in the EXTPROC keyword of the prototype, or of the procedure interface, if the prototype is not specified.

Any procedure interface found in the global definitions is assumed to be the procedure interface for the cycle-main procedure. If a prototype is specified, the name is required for the procedure interface for the cycle-main procedure, and the prototype with the matching name must precede the procedure interface in the source.

In the following example, module CheckFile is created. Its cycle-main procedure has three parameters:

1. A file name (input)
2. A library name (input)
3. An indicator indicating whether the file was found (output)

In this example, the procedure is intended to be called from another module, so a prototype must be specified in a /COPY file.

/COPY file CHECKFILEC with the prototype for the cycle-main procedure:

```
D CheckFile       PR
D   file                        10a   const
D   library                     10a   const
D   found                        1N
```

Module CheckFile:

```
 /COPY CHECKFILEC
D CheckFile       PI
D   file                        10a   const
D   library                     10a   const
D   found                        1N
C   ... code using parameters file, library and found
```

Using a *ENTRY PLIST, you would define the parameters this way:

```
D file            S             10a   const
D library         S             10a   const
D found           S              1N
C     *ENTRY        PLIST
C                   PARM                    file
C                   PARM                    library
C                   PARM                    found
C   ... code using parameters file, library and found
```

You can also use a prototype and procedure interface to define your cycle-main procedure as a program. In this case, you would specify the EXTPGM keyword for the prototype. In this example, the program is intended to be called by other RPG programs, so a prototype must be specified in a /COPY file.

/COPY file CHECKFILEC with the prototype for the program:

```
 D CheckFile       PR                  extpgm('CHECKFILE')
 D   file                        10a   const
 D   library                     10a   const
 D   found                        1N
```

In the module source, the procedure interface would be defined the same way.

In the following example, the program is not intended to be called by any other RPG programs, so a prototype is not necessary. In this case, the EXTPGM keyword is specified for the procedure interface. Since a prototype is not specified, a name is not necessary for the procedure interface.

A procedure interface with the EXTPGM keyword:

```
 F ... file specifications
 D                PI                   extpgm('CUSTREPORT')
 D   custfile                    10a   const
 D   custlib                     10a   const
  ... code using the custfile and custlib parameters
```

Use Caution Exporting Subprocedures in Cycle Modules

If a module contains both a cycle-main procedure and exported subprocedures, take great care to ensure that the RPG cycle in the cycle-main procedure does not adversely affect the global data, files, and data areas that the subprocedures are using.

You must be aware of when files are opened and closed implicitly, when data areas are locked and unlocked implicitly, and when global data is initialized or re-initialized.

Potential Problem Situations: A cycle module having exported subprocedures introduces potential scenarios where the cycle-main procedure initialization is performed at an unexpected time, with the effect that has on files, data area locks, and global data then leading to errors. An exported subprocedure can be called first in the module, from a procedure outside the module, before the cycle-main procedure is called. If the cycle-main procedure is then called, it will initialize at that time.

- If module initialization occurs because a subprocedure is the first procedure to be called, and cycle-main procedure initialization occurs later, errors can occur if files are already open or data areas are already locked.
- If a subprocedure calls the cycle-main procedure, global data may or may not be reinitialized during the call, depending on the way the main procedure ended the last time it was called. If the subprocedure is using any global data, this can cause unexpected results.
- If the cycle-main procedure was last called and ended and implicitly closed the files and unlocked the data areas, and an exported subroutine is then called from outside the module, errors can occur if it expects those files to be open or data areas to be locked.

Recommendations: Consider moving the cycle-main procedure logic into a subprocedure, and making the module a NOMAIN module, or changing the cycle-main procedure to be a linear-main procedure.

If you mix cycle-main procedures with exported subprocedures, ensure that your cycle-main procedure is called first, before any subprocedures.

Do not allow cycle-main-procedure initialization to happen more than once, since this would reinitialize your global data. The best way to prevent reinitialization is to avoid using the LR indicator.

If you want to call your cycle-main procedure intermixed with your subprocedures, you should declare all your files as USROPN and not use UDS data areas. Open files and lock data areas as you need them, and close files and unlock data areas when you no longer need them. You might consider having a subprocedure in the module that will close any open files and unlock any locked data areas.

Linear Module

A module which specifies the **MAIN** or **NOMAIN** keyword on the Control specification is compiled without incorporating the program cycle.

When the program cycle is not included in the module, you are restricted in terms of what can be coded in the main source section. Specifically, you cannot code specifications for:

- Primary and secondary files
- Heading, detail and total output
- Executable calculations, including the *INZSR Initialization subroutine
- *ENTRY PLIST

Instead you would code in the main source section:

- Full-procedural files
- Input specifications
- Definition specifications
- Declarative calculations such as DEFINE, KFLD, KLIST, PARM, and PLIST (but not *ENTRY PLIST)
- Exception output

Caution: There is no implicit closing of global files or unlocking of data areas in a linear module. These objects will remain open or locked until they are explicitly closed or unlocked.

Linear Main Module

A module which has a program entry procedure but does not use the RPG Program Cycle can be generated by specifying the MAIN keyword on the control specification.

This type of module has one or more procedures, one of which is identified as the main procedure. It does not allow specifications which relate to the RPG Program Cycle.

See “MAIN(main_procedure_name)” on page 269 for more information.

NOMAIN Module

You can code one or more subprocedures in a module without coding a main procedure. Such a module is called a **NOMAIN module**, since it requires the specification of the NOMAIN keyword on the control specification. No cycle code is generated for the NOMAIN module.

> **TIP**
>
> You may want to consider converting all your Cycle modules to NOMAIN modules except the ones that actually contain the program entry procedure for a program, to reduce the individual size of those modules by eliminating the unnecessary cycle code in each of those modules.

Note: A module with NOMAIN specified will not have a program entry procedure. Consequently you cannot use the CRTBNDRPG command to compile the source.

See “NOMAIN” on page 271 for more information.

Module Initialization

Module initialization occurs when the first procedure (either the main procedure or a subprocedure) is called.

A cycle module has an additional form of initialization which can occur repeatedly. Cycle-main procedure initialization occurs when the cycle-main procedure is called the first time. It also occurs on subsequent calls if the cycle-main procedure ended abnormally or with LR on.

Initialization of Global Data

Global data in the module is initialized during module initialization and during cycle-main procedure initialization.

For special concerns regarding initialization in cycle-main procedures, see "Use
Caution Exporting Subprocedures in Cycle Modules" on page 28.

RPG Cycle and other implicit Logic

The ILE RPG compiler supplies part of the logic for an RPG program.

- For a cycle-main procedure, the compiler supplies the program cycle; the program cycle is also called the *logic cycle* or the *RPG cycle*
- For a subprocedure or linear-main procedure, the compiler supplies the initialization and termination of the subprocedure.

Program Cycle

The ILE RPG compiler supplies part of the logic for an RPG program. For a cycle-main procedure, the logic the compiler supplies is called the program cycle or logic cycle. The program cycle is a series of ordered steps that the main procedure goes through for each record read.

The information that you code on RPG IV specifications in your source program need not explicitly specify when records should be read or written. The ILE RPG compiler can supply the logical order for these operations when your source program is compiled. Depending on the specifications you code, your program may or may not use each step in the cycle.

Primary (identified by a P in position 18 of the file description specifications) and secondary (identified by an S in position 18 of the file description specifications) files indicate input is controlled by the program cycle. A full procedural file (identified by an F in position 18 of the file description specifications) indicates that input is controlled by program-specified calculation operations (for example, READ and CHAIN).

To control the cycle, you can have:

- One primary file and, optionally, one or more secondary files
- Only full procedural files
- A combination of one primary file, optional secondary files, and one or more full procedural files in which some of the input is controlled by the cycle, and other input is controlled by the program.
- No files (for example, input can come from a parameter list or a data area data structure).

Note: No cycle code is generated for a module when MAIN or NOMAIN is specified on the control specification. See "Linear Module" on page 29 for more information.

General RPG IV Program Cycle

Figure 7 on page 32 shows the specific steps in the general flow of the RPG IV program cycle. A program cycle begins with step 1 and continues through step 7, then begins again with step 1.

The first and last time a program goes through the RPG IV cycle differ somewhat from the normal cycle. Before the first record is read the first time through the cycle, the program resolves any parameters passed to it, writes the records conditioned by the 1P (first page) indicator, does file and data initialization, and processes any heading or detail output operations having no conditioning indicators or all negative conditioning indicators. For example, heading lines printed before the first record is read might consist of constant or page heading information or fields for reserved words, such as PAGE and *DATE. In addition, the program bypasses total calculations and total output steps on the first cycle.

During the last time a program goes through the cycle, when no more records are available, the LR (last record) indicator and L1 through L9 (control level) indicators are set on, and file and data area cleanup is done.

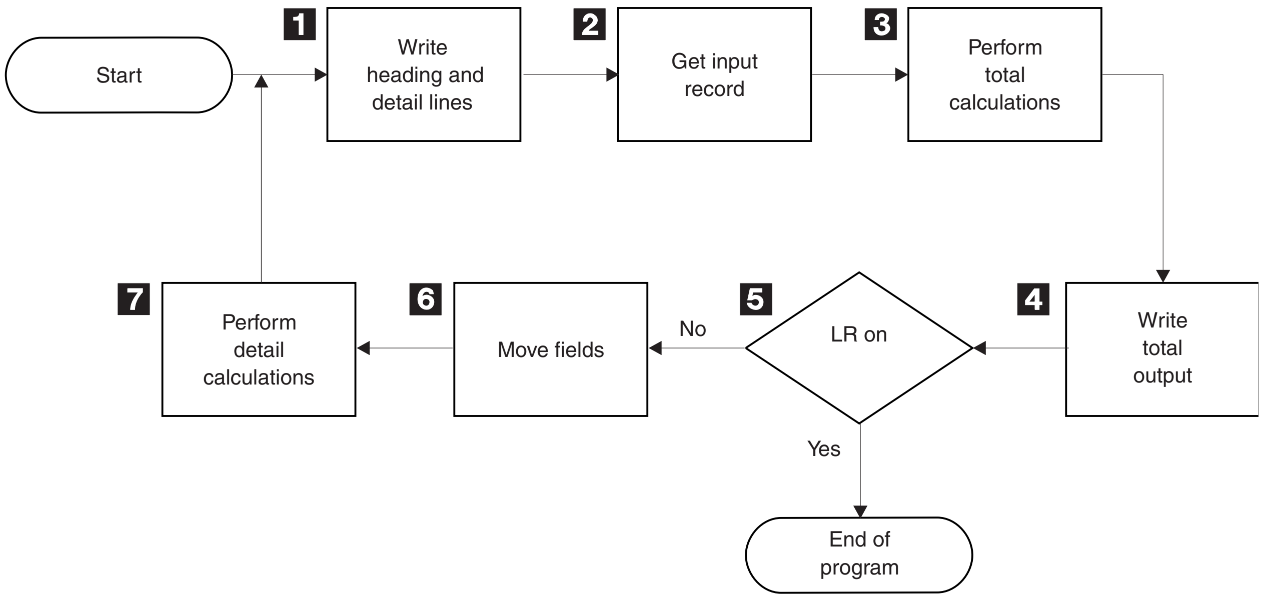


Figure 7. RPG IV Program Logic Cycle

1 All heading and detail lines (H or D in position 17 of the output specifications) are processed.

2 The next input record is read and the record identifying and control level indicators are set on.

3 Total calculations are processed. They are conditioned by an L1 through L9 or LR indicator, or an L0 entry.

4 All total output lines are processed. (identified by a T in position 17 of the output specifications).

5 It is determined if the LR indicator is on. If it is on, the program is ended.

6 The fields of the selected input records are moved from the record to a processing area. Field indicators are set on.

7 All detail calculations are processed (those not conditioned by control level indicators in positions 7 and 8 of the calculation specifications) on the data from the record read at the beginning of the cycle.

Detailed RPG IV Program Cycle

In “General RPG IV Program Cycle” on page 31, the basic RPG IV Logic Cycle was introduced. The following figures provide a detailed explanation of the RPG IV Logic Cycle.

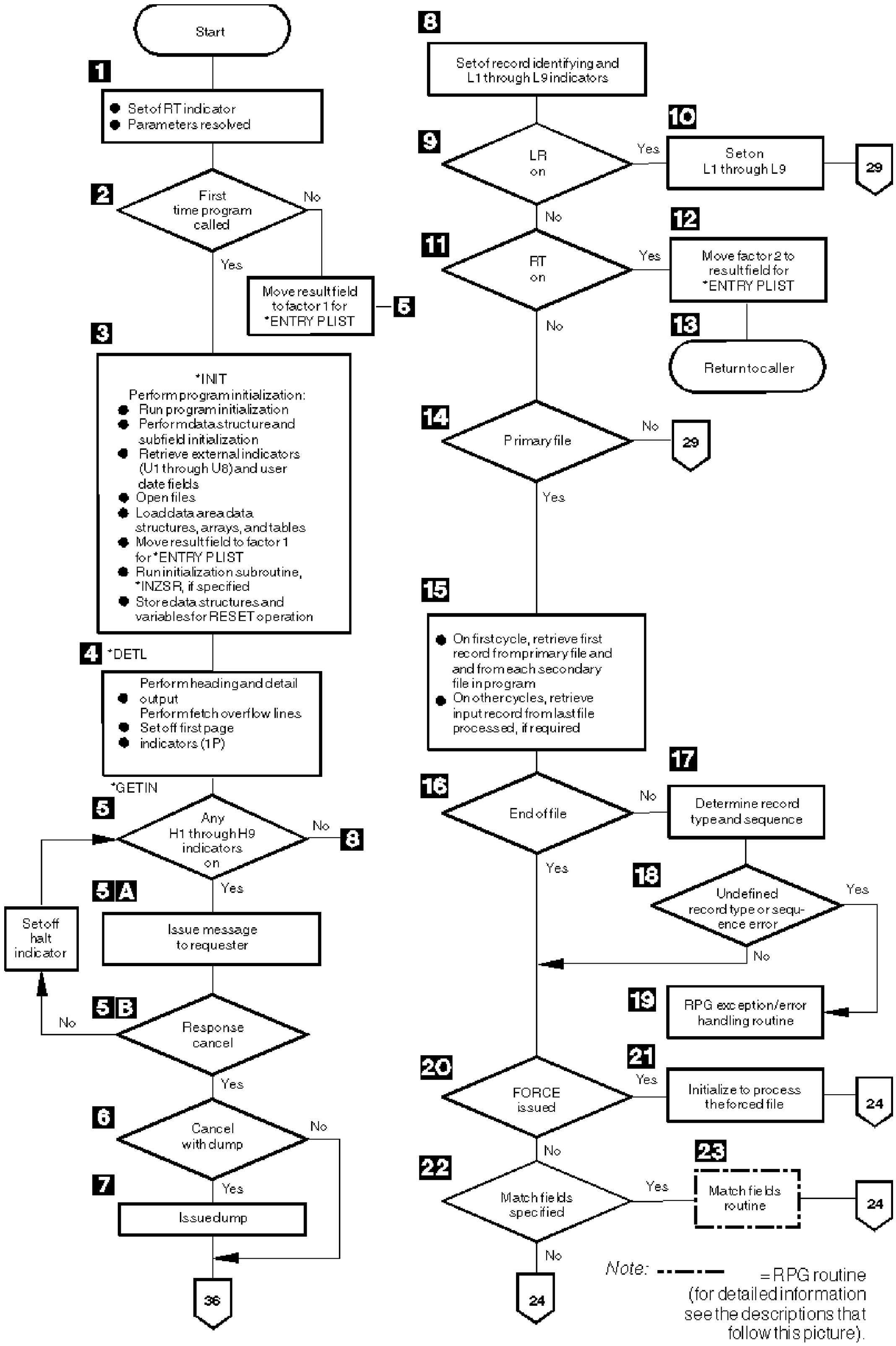


Figure 8. Detailed RPG IV Object Program Cycle (Part 1 of 2)

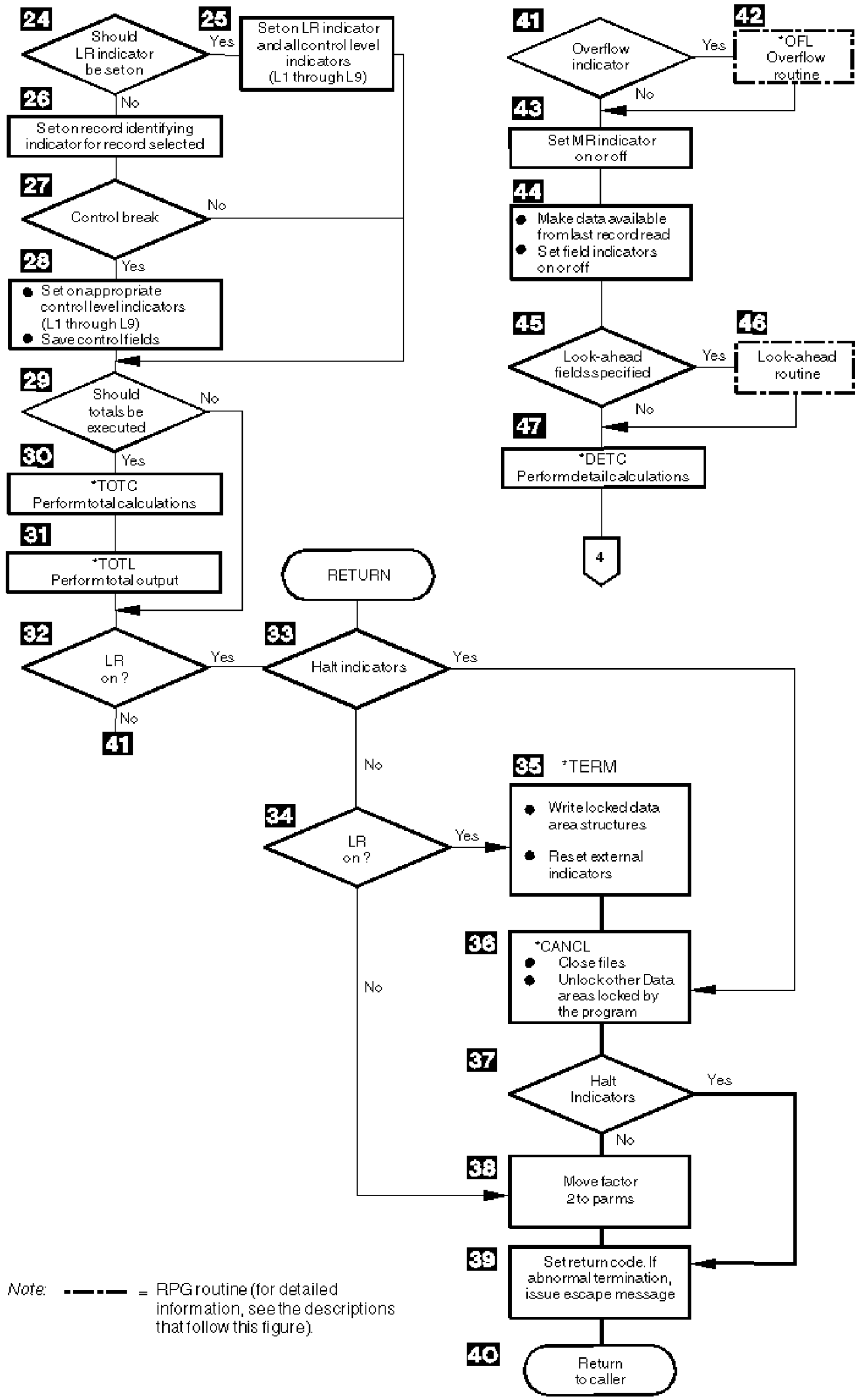


Figure 8. Detailed RPG IV Object Program Cycle (Part 2 of 2)

Detailed RPG IV Object Program Cycle: Figure 8 on page 33 shows the specific steps in the detailed flow of the RPG IV program cycle. The item numbers in the following description refer to the numbers in the figure. Routines are flowcharted in Figure 11 on page 42 and in Figure 9 on page 39.

1 The RT indicator is set off. If *ENTRY PLIST is specified the parameters are resolved.

2 RPG IV checks for the first invocation of the program. If it is the first invocation, program initialization continues. If not, it moves the result field to factor 1 in the PARM statements in *ENTRY PLIST and branches to step 5.

3 The program is initialized at *INIT in the cycle. This process includes: performing data structure and subfield initialization, setting user date fields; opening global files; loading all data area data structures, arrays and tables; moving the result field to factor 1 in the PARM statements in *ENTRY PLIST; running the initialization subroutine *INZSR; and storing the structures and variables for the RESET operation. Global files are opened in reverse order of their specification on the File Description Specifications.

4 Heading and detail lines (identified by an H or D in position 17 of the output specifications) are written before the first record is read. Heading and detail lines are always processed at the same time. If conditioning indicators are specified, the proper indicator setting must be satisfied. If fetch overflow logic is specified and the overflow indicator is on, the appropriate overflow lines are written. File translation, if specified, is done for heading and detail lines and overflow output. This step is the return point in the program if factor 2 of an ENDSR operation contains the value *DETL.

5 The halt indicators (H1 through H9) are tested. If all the halt indicators are off, the program branches to step 8. Halt indicators can be set on anytime during the program. This step is the return point in the program if factor 2 of an ENDSR operation contains the value *GETIN.

a. If any halt indicators are on, a message is issued to the user.

b. If the response is to continue, the halt indicator is set off, and the program returns to step 5. If the response is to cancel, the program goes to step 6.

6 If the response is to cancel with a dump, the program goes to step 7; otherwise, the program branches to step 36.

7 The program issues a dump and branches to step 36 (abnormal ending).

8 All record identifying, 1P (first page), and control level (L1 through L9) indicators are set off. All overflow indicators (OA through OG, OV) are set off unless they have been set on during preceding detail calculations or detail output. Any other indicators that are on remain on.

9 If the LR (last record) indicator is on, the program continues with step 10. If it is not on, the program branches to step 11.

10 The appropriate control level (L1 through L9) indicators are set on and the program branches to step 29.

11 If the RT indicator is on, the program continues with step 12; otherwise, the program branches to step 14.

12 Factor 2 is moved to the result field for the parameters of the *ENTRY PLIST.

13 If the RT indicator is on (return code set to 0), the program returns to the caller.

14 If a primary file is present in the program, the program continues with step 15; otherwise, the program branches to step 29.

15 During the first program cycle, the first record from the primary file and from each secondary file in the program is read. File translation is done on the input records. In other program cycles, a record is read from the last file processed. If this file is processed by a record address file, the data in the record address file defines the record to be retrieved. If lookahead fields are specified in the last record processed, the record may already be in storage; therefore, no read may be done at this time.

16 If end of file has occurred on the file just read, the program branches to step 20. Otherwise, the program continues with step 17.

17 If a record has been read from the file, the record type and record sequence (positions 17 through 20 of the input specifications) are determined.

18 It is determined whether the record type is defined in the program, and if the record sequence is correct. If the record type is undefined or the record sequence is incorrect, the program continues with step 19; otherwise, the program branches to step 20.

19 The RPG IV exception/error handling routine receives control.

20 It is determined whether a FORCE operation was processed on the previous cycle. If a FORCE operation was processed, the program selects that file for processing (step 21) and branches around the processing for match fields (steps 22 and 23). The branch is processed because all records processed with a FORCE operation are processed with the matching record (MR) indicator off.

21 If FORCE was issued on the previous cycle, the program selects the forced file for processing after saving any match fields from the file just read. If the file forced is at end of file, normal primary/secondary multifile logic selects the next record for processing and the program branches to step 24.

22 If match fields are specified, the program continues with step 23; otherwise, the program branches to step 24.

23 The match fields routine receives control. (For detailed information on the match fields routine, see “Match Fields Routine” on page 39.)

24 The LR (last record) indicator is set on when all records are processed from the files that have an E specified in position 19 of the file description specifications and all matching secondary records have been processed. If the LR indicator is not set on, processing continues with step 26.

25 The LR (last record) indicator is set on and all control level (L1 through L9) indicators, and processing continues with step 29.

26 The record identifying indicator is set on for the record selected for processing.

27 It is determined whether the record selected for processing caused a control break. A control break occurs when the value in the control fields of the record being processed differs from the value of the control fields of the last record processed. If a control break has not occurred, the program branches to step 29.

28 When a control break occurs, the appropriate control level indicator (L1 through L9) is set on. All lower level control indicators are set on. The program saves the contents of the control fields for the next comparison.

29 It is determined whether the total-time calculations and total-time output should be done. Totals are always processed when the LR indicator is on. If no control level is specified on the input specifications, totals are bypassed on the first cycle and after the first cycle, totals are processed on every cycle. If control levels are specified on the input specifications, totals are bypassed until after the first record containing control fields has been processed.

30 All total calculations conditioned by a control level entry (positions 7 and 8 of the calculation specifications). are processed. This step is the return point in the program if factor 2 of an ENDSR operation contains the value *TOTC.

31 All total output is processed. If fetch overflow logic is specified and the overflow indicator (OA through OG, OV) associated with the file is on, the overflow lines are written. File translation, if specified, is done for all total output and overflow lines. This step is the return point in the program if factor 2 of an ENDSR operation contains the value *TOTL.

32 If LR is on, the program continues with step 33; otherwise, the program branches to step 41.

33 The halt indicators (H1 through H9) are tested. If any halt indicators are on, the program branches to step 36 (abnormal ending). If the halt indicators are off, the program continues with step 34. If the RETURN operation code is used in calculations, the program branches to step 33 after processing of that operation.

34 If LR is on, the program continues with step 35. If it is not on, the program branches to step 38.

35 RPG IV program writes all arrays or tables for which the TOFILE keyword has been specified on the definition specification and writes all locked data area data structures. Output arrays and tables are translated, if necessary.

36 All open global files are closed. The RPG IV program also unlocks all data areas that have been locked but not unlocked by the program. If factor 2 of an ENDSR operation contains the value *CANCL, this step is the return point.

37 The halt indicators (H1 through H9) are tested. If any halt indicators are on, the program branches to step 39 (abnormal ending). If the halt indicators are off, the program continues with step 38.

38 The factor 2 fields are moved to the result fields on the PARMs of the *ENTRY PLIST.

39 The return code is set. 1 = LR on, 2 = error, 3 = halt.

40 Control is returned to the caller.

Note: Steps 32 through 40 constitute the normal ending routine. For an abnormal ending, steps 34 through 35 are bypassed.

41 It is determined whether any overflow indicators (OA through OG OV) are on. If an overflow indicator is on, the program continues with step 42; otherwise, the program branches to step 43.

42 The overflow routine receives control. (For detailed information on the overflow routine, see "Overflow Routine" on page 39.) This step is the return point in the program if factor 2 of an ENDSR operation contains the value *OFL.

43 The MR indicator is set on and remains on for the complete cycle that processes the matching record if this is a multifile program and if the record to be processed is a matching record. Otherwise, the MR indicator is set off.

44 Data from the last record read is made available for processing. Field indicators are set on, if specified.

45 If lookahead fields are specified, the program continues with step 46; otherwise, the program branches to step 47.

46 The lookahead routine receives control. (For detailed information on the lookahead routine, see "Lookahead Routine" on page 40.)

47 Detail calculations are processed. This step is the return point in the program if factor 2 of an ENDSR operation contains the value *DETC. The program branches to step 4.

Initialization Subroutine: Refer to Figure 8 on page 33 to see a detailed explanation of the RPG IV initialization subroutine.

The initialization subroutine allows you to process calculation specifications before 1P output. A specific subroutine that is to be run at program initialization time can be defined by specifying *INZSR in factor 1 of the subroutine's BEGSR operation. Only one subroutine can be defined as an initialization subroutine. It is called at the end of the program initialization step of the program cycle (that is, after data structures and subfields are initialized, external indicators and user data fields are retrieved, global files are opened, data area data structures, arrays, and tables are loaded, and PARM result fields moved to factor 1 for *ENTRY PLIST). *INZSR may not be specified as a file/program error/exception subroutine.

If a program ends with LR off, the initialization subroutine does not automatically run during the next invocation of that program because the subroutine is part of the initialization step of the program. However, if the initialization subroutine does not complete before an exit is made from the program with LR off, the initialization subroutine will be re-run at the next invocation of that program.

The initialization subroutine is like any other subroutine in the program, other than being called at program initialization time. It may be called using the EXSR or CASxx operations, and it may call other subroutines or other programs. Any operation that is valid in a subroutine is valid in the initialization subroutine, with the exception of the RESET operation. This is because the value used to reset a variable is not defined until after the initialization subroutine is run.

Any changes made to a variable during the initialization subroutine affect the value that the variable is set to on a subsequent RESET operation. Default values can be defined for fields in record formats by, for example, setting them in the initialization subroutine and then using RESET against the record format whenever the default values are to be used. The initialization subroutine can also retrieve information such as the current time for 1P output.

There is no *INZSR associated with subprocedures. If a subprocedure is the first procedure called in a module, the *INZSR of the main procedure will not be run, although other initialization of global data will be done. The *INZSR of the main procedure will be run when the main procedure is called.

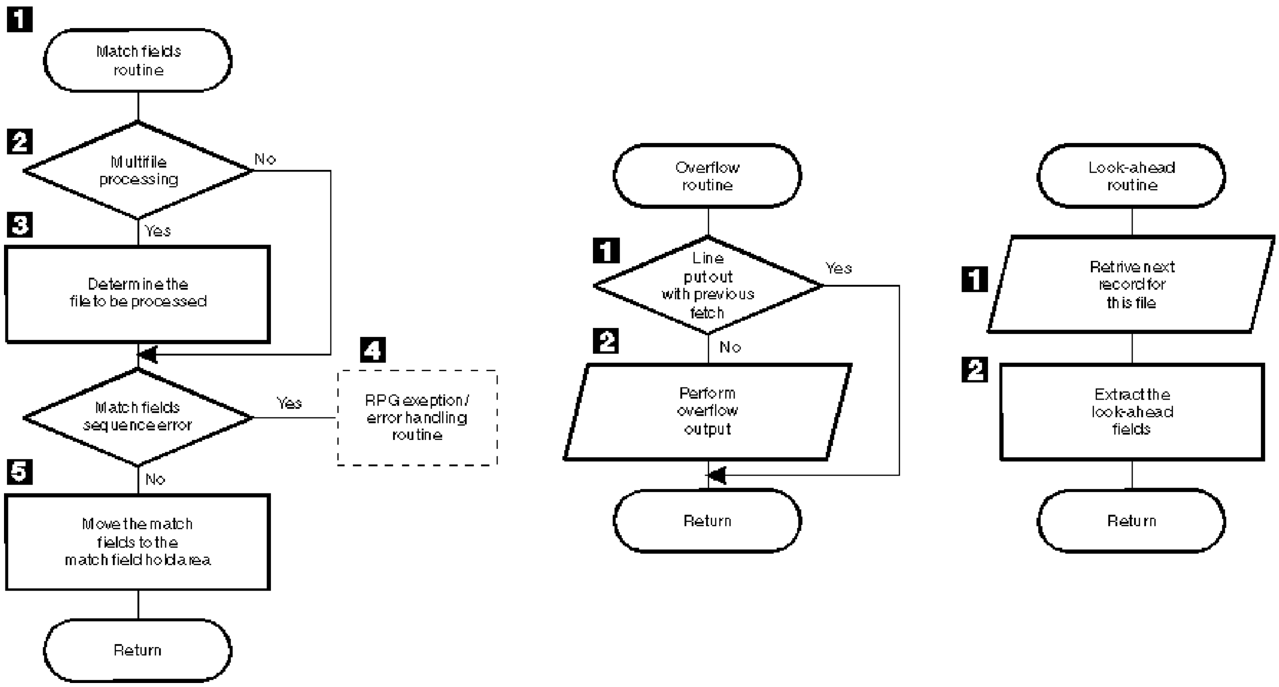


Figure 9. Detail Flow of RPG IV Match Fields, Overflow, and Lookahead Routines

Match Fields Routine: Figure 9 shows the specific steps in the RPG IV match fields routine. The item numbers in the following descriptions refer to the numbers in the figure.

1 If multifile processing is being used, processing continues with step 2; otherwise, the program branches to step 3.

2 The value of the match fields in the hold area is tested to determine which file is to be processed next.

3 The RPG IV program extracts the match fields from the match files and processes sequence checking. If the match fields are in sequence, the program branches to step 5.

4 If the match fields are not in sequence, the RPG IV exception/error handling routine receives control.

5 The match fields are moved to the hold area for that file. A hold area is provided for each file that has match fields. The next record is selected for processing based on the value in the match fields.

Overflow Routine: Figure 9 shows the specific steps in the RPG IV overflow routine. The item numbers in the following descriptions refer to the numbers in the figure.

1 The RPG IV program determines whether the overflow lines were written previously using the fetch overflow logic (step 30 in Figure 8 on page 33). If the overflow lines were written previously, the program branches to the specified return point; otherwise, processing continues with step 2.

2 All output lines conditioned with an overflow indicator are tested and written to the conditioned overflow lines.

The fetch overflow routine allows you to alter the basic RPG IV overflow logic to prevent printing over the perforation and to let you use as much of the page as possible. During the regular program cycle, the RPG IV program checks only once, immediately after total output, to see if the overflow indicator is on. When the fetch overflow function is specified, the RPG IV program checks overflow on each line for which fetch overflow is specified.

Specify fetch overflow with an F in position 18 of the output specifications on any detail, total, or exception lines for a PRINTER file. The fetch overflow routine does not automatically cause forms to advance to the next page.

During output, the conditioning indicators on an output line are tested to determine whether the line is to be written. If the line is to be written and an F is specified in position 18, the RPG IV program tests to determine whether the overflow indicator is on. If the overflow indicator is on, the overflow routine is fetched and the following operations occur:

- Only the overflow lines for the file with the fetch specified are checked for output.
- All total lines conditioned by the overflow indicator are written.
- Forms advance to a new page when a skip to a line number less than the line number the printer is currently on is specified in a line conditioned by an overflow indicator.
- Heading, detail, and exception lines conditioned by the overflow indicator are written.
- The line that fetched the overflow routine is written.
- Any detail and total lines left to be written for that program cycle are written.

Position 18 of each OR line must contain an F if the overflow routine is to be used for each record in the OR relationship. Fetch overflow cannot be used if an overflow indicator is specified in positions 21 through 29 of the same specification line. If this occurs, the overflow routine is not fetched.

Use the fetch overflow routine when there is not enough space left on the page to print the remaining detail, total, exception, and heading lines conditioned by the overflow indicator. To determine when to fetch the overflow routine, study all possible overflow situations. By counting lines and spaces, you can calculate what happens if overflow occurs on each detail, total, and exception line.

Lookahead Routine: Figure 9 on page 39 shows the specific steps in the RPG IV lookahead routine. The item numbers in the following descriptions refer to the numbers in the figure.

1 The next record for the file being processed is read. However, if the file is a combined or update file (identified by a C or U, respectively, in position 17 of the file description specifications), the lookahead fields from the current record being processed is extracted.

2 The lookahead fields are extracted.

Ending a Program without a Primary File: If your program does not contain a primary file, you *must* specify a way for the program to end:

- By setting the LR indicator on
- By setting the RT indicator on
- By setting an H1 through H9 indicator on
- By specifying the RETURN operation code

The LR, RT, H1 through H9 indicators, and the RETURN operation code, can be used in conjunction with each other.

Program Control of File Processing: Specify a full procedural file (F in position 18 of the file description specifications) to control all or partial input of a program. A full procedural file indicates that *input* is controlled by program-specified calculation operations (for example, READ, CHAIN). When both full procedural files and a primary file (P in position 18 of the file description specifications) are specified in a program, some of the input is controlled by the program, and other input is controlled by the cycle. Even if the program cycle exists in your module, all the processing of a full-procedural file is done in your calculations.

The file operation codes can be used for program control of input. These file operation codes are discussed in "File Operations" on page 453.

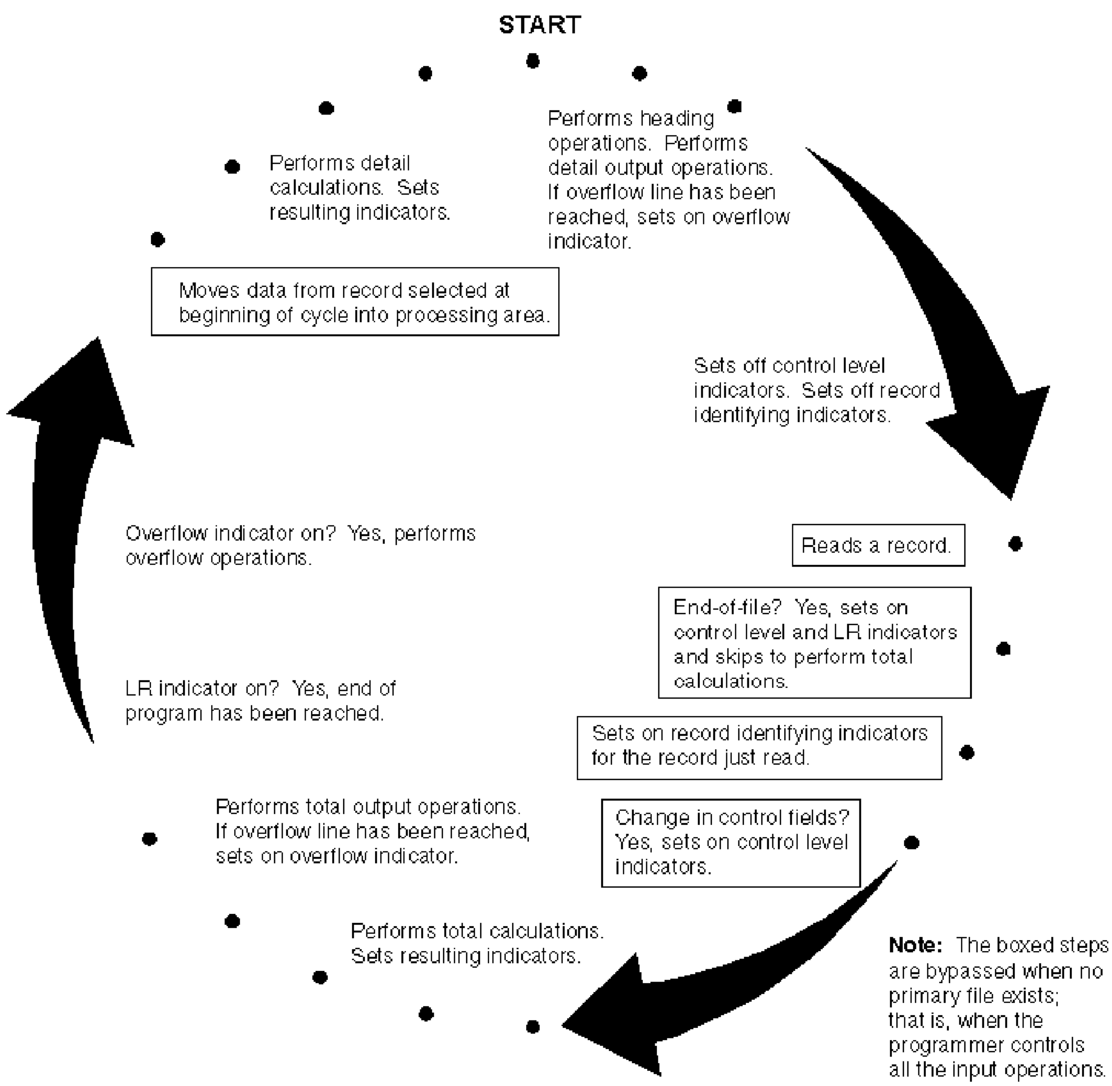


Figure 10. Programmer Control of Input Operation within the Program-Cycle

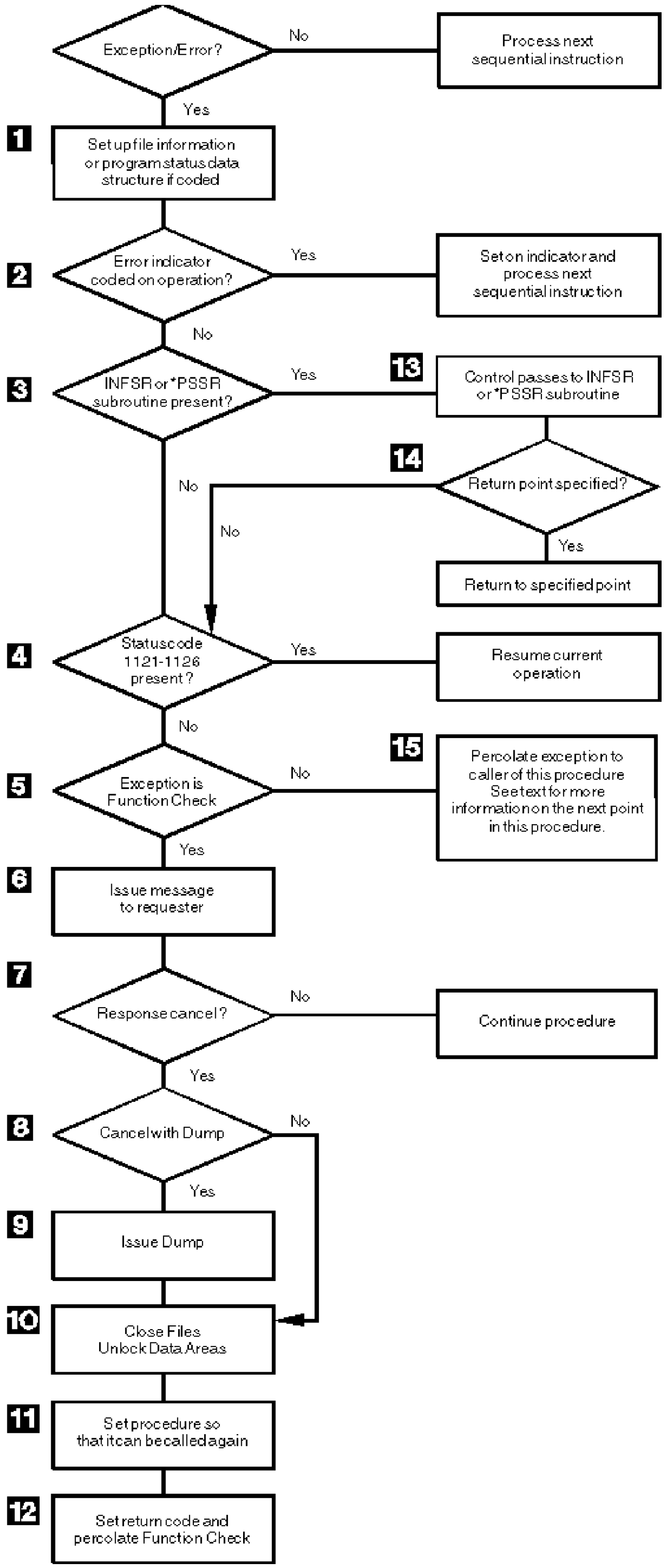


Figure 11. Detail Flow of RPG IV Exception/Error Handling Routine

RPG IV Exception/Error Handling Routine: Figure 11 shows the specific steps in the RPG IV exception/error handling routine. The item numbers in the following description refer to the numbers in the figure.

1 Set up the file information or procedure status data structure, if specified, with status information.

2 If the exception/error occurred on an operation code that has an indicator specified in positions 73 and 74, the indicator is set on, and control returns to the next sequential instruction in the calculations.

3 If the appropriate exception/error subroutine (INFSR or *PSSR) is present in the procedure, the procedure branches to step 13; otherwise, the procedure continues with step 4.

4 If the Status code is 1121-1126 (see “File Status Codes” on page 91), control returns to the current instruction in the calculations. If not, the procedure continues with step 5.

5 If the exception is a function check, the procedure continues with step 6. If not, it branches to step 15.

6 An inquiry message is issued to the requester. For an interactive job, the message goes to the requester. For a batch job, the message goes to QSYSOPR. If QSYSOPR is not in break mode, a default response is issued.

7 If the user's response is to cancel the procedure, the procedure continues with step 8. If not, the procedure continues.

8 If the user's response is to cancel with a dump, the procedure continues with step 9. If not, the procedure branches to step 10.

9 A dump is issued.

10 All global files are closed and data areas are unlocked

11 The procedure is set so that it can be called again.

12 The return code is set and the function check is percolated.

13 Control passes to the exception/error subroutine (INFSR or *PSSR).

14 If a return point is specified in factor 2 of the ENDSR operation for the exception/error subroutine, the procedure goes to the specified return point. If a return point is not specified, the procedure goes to step 4. If a field name is specified in factor 2 of the ENDSR operation and the content is not one of the RPG IV-defined return points (such as *GETIN or *DETC), the procedure goes to step 6. No error is indicated, and the original error is handled as though the factor 2 entry were blank.

15 If no invocation handles the exception, then it is promoted to function check and the procedure branches to step 5. Otherwise, depending on the action taken by the handler, control resumes in this procedure either at step 10 or at the next machine instruction after the point at which the exception occurred.

Subprocedure Calculations

No cycle code is generated for a subprocedure, and so you must code it differently than you would code a cycle-main procedure. The 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 12 on page 44 shows the normal processing steps for a subprocedure. Figure 13 on page 45 shows the exception/error handling sequence.

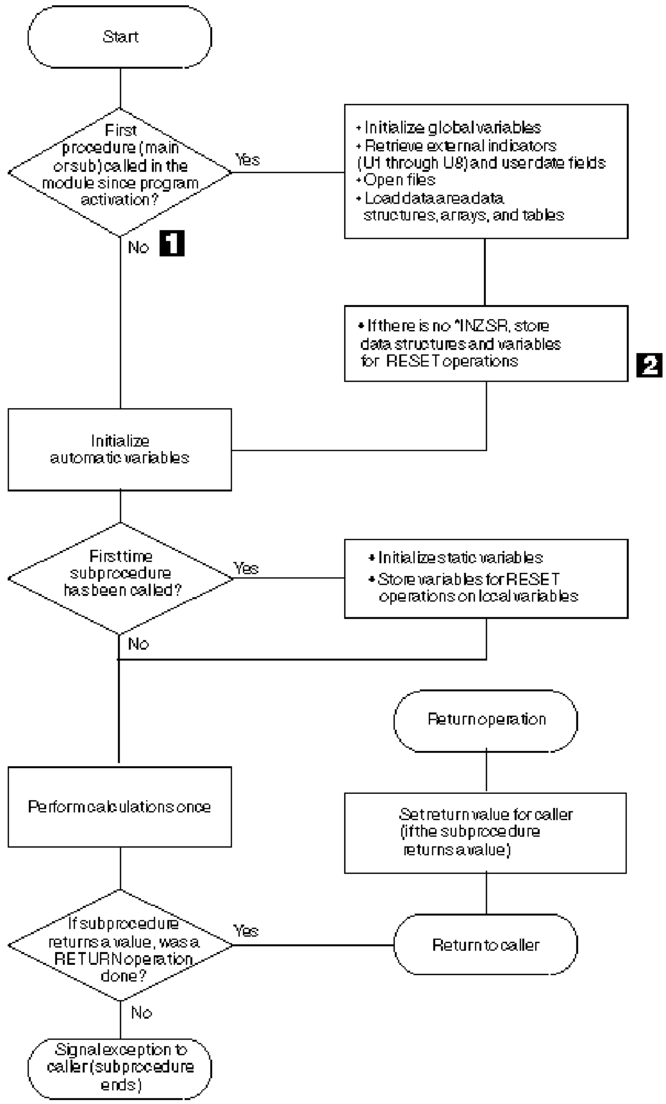


Figure 12. Normal Processing Sequence for a Subprocedure

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.

2 If an entry parameter to the main procedure is RESET anywhere in the
module, this will cause an exception. If it is possible that a subprocedure
will be called before the main procedure, it is not advised to RESET any
entry parameters for the cycle-main procedure.

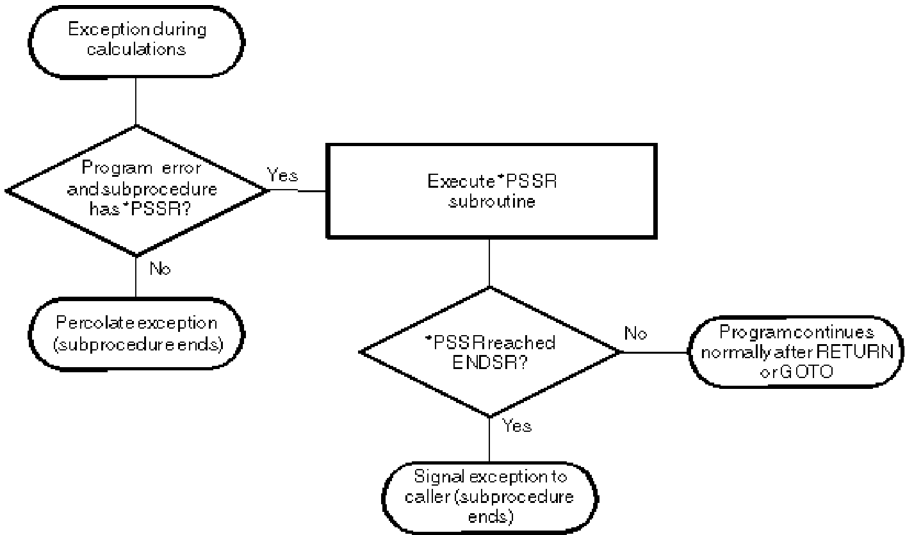


Figure 13. Exception/Error Handling Sequence for a Subprocedure

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.

  Note also that if the subprocedure is the *first* procedure to be called in a module, the *INZSR of the cycle-main procedure (if present) will not be run, although other initialization of global data will be done. The *INZSR of the cycle-main procedure will be run when the cycle-main procedure is called.
- 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. You can set on the LR or RT indicators, but it will have no immediate effect on the program termination. If the the subprocedure was called by a cycle-main procedure, the setting of the LR or RT indicators would take effect when the RPG cycle reached the point at which RPG checks those indicators.
- Exception handling within a subprocedure differs from a cycle-main procedure primarily because there is no default exception handler for subprocedures and so situations where the default handler would be called for a cycle-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 of the ENDSR for the *PSSR subroutine in a cycle-main procedure would result in control being passed to the default handler. In a subprocedure, if the ENDSR of the *PSSR subroutine is reached, then the subprocedure will end abnormally and RNX9001 will be signalled to the caller of the subprocedure.

  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.
- You cannot code an INFSR in a subprocedure, nor can you use a file for which an INFSR is coded.
- Indicators that control the cycle function solely as conditioning indicators when used in a linear module (MAIN or NOMAIN on control specification); or in a

subprocedure that is active, but where the cycle-main procedure of the module is not. Indicators that control the cycle include: LR, RT, H1-H9, and control level indicators.

Implicit Opening of Files and Locking of Data Areas

UDS data areas and global files that do not have the USROPN keyword are opened or locked implicitly during module initialization and during cycle-main-procedure initialization. Static files in subprocedures that do not have the USROPN keyword are opened implicitly the first time the subprocedure is called. Automatic files in subprocedures that do not have the USROPN keyword are opened every time the procedure is called.

Implicit Closing of Files and Unlocking of Data Areas

Global files that are open are closed implicitly, and data areas that are locked are unlocked implicitly during cycle-main procedure termination, when the cycle-main procedure ends abnormally or with LR on. Automatic files in subprocedures are closed implicitly when the subprocedure ends normally or abnormally.

Caution: There is no implicit closing of static files in subprocedures. There is no closing of global files or implicit unlocking of data areas in a linear module. These objects will remain open or locked unless they are explicitly closed or unlocked.

Chapter 4. RPG IV Indicators

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

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 Format” on page 182 and “Position 40 (Internal Data Type)” on page 320. This chapter describes a special set of predefined RPG IV indicators (*INxx).

RPG IV indicators are defined either by an entry on a specification or by the RPG IV program itself. The positions on the specification in which you define the indicator determine how the indicator is used. An indicator that has been defined can then be used to condition calculation and output operations.

The RPG IV program sets and resets certain indicators at specific times during the program cycle. In addition, the state of most indicators can be changed by calculation operations. All indicators except MR, 1P, KA through KN, and KP through KY can be set on with the SETON operation code; all indicators except MR and 1P can be set off with the SETOFF operation code.

This chapter is divided into the following topics:
- Indicators defined on the RPG IV specifications
- Indicators not defined on the RPG IV specifications
- Using indicators
- Indicators referred to as data.

Indicators Defined on RPG IV Specifications

You can specify the following indicators on the RPG IV specifications:
- Overflow indicator (the OFLIND keyword on the file description specifications).
- Record identifying indicator (positions 21 and 22 of the input specifications).
- Control level indicator (positions 63 and 64 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 (See “Indicators Referred to As Data” on page 73 for a description of how an indicator is defined when used with one of these reserved words.).

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

Overflow Indicators

An overflow indicator is defined by the OFLIND keyword on the file description specifications. It is set on when the last line on a page has been printed or passed. Valid indicators are *INOA through *INOG, *INOV, and *IN01 through *IN99. A defined overflow indicator can then be used to condition calculation and output operations. A description of the overflow indicator and fetch overflow logic is given in “Overflow Routine” on page 39.

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 valid record identifying indicators are:
- 01-99
- H1-H9
- L1-L9
- LR
- U1-U8
- RT

For an externally described file, a record identifying indicator is optional, but, if you specify it, it follows the same rules as for a program described file.

Generally, the indicators 01 through 99 are used as record identifying indicators. However, the control level indicators (L1 through L9) and the last record indicator (LR) can be used. If L1 through L9 are specified as record identifying indicators, lower level indicators are not set on.

When you select a record type 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 at detail and total calculation time to retrieve records from a file (see below). The record identifying indicator is set on after the record is selected, but before the input fields are moved to the input area. The record identifying indicator for the new record is on during total time for the old record; therefore, calculations processed at total time using the fields of the old record cannot be conditioned by the record identifying indicator of the old record. You can set the indicators off at any time in the program cycle; they are set off before the next primary or secondary record is selected.

If you use a file operation code 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. The record identifying indicator is not set off until the appropriate point in the RPG IV cycle. (See Figure 10 on page 41.) Therefore, 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 within the same RPG IV program cycle.

Rules for Assigning Record Identifying Indicators

When you assign record identifying indicators to records in a program described file, remember the following:
- You can assign the same indicator to two or more different record types if the same operation is to be processed on all record types. To do this, you 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.
- You can associate a record identifying indicator 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.

When you assign record identifying indicators to records in an externally described file, remember the following:

- 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.

For an example of record identifying indicators, see Figure 14.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
 *
I*Record identifying indicator 01 is set on if the record read
I*contains an S in position 1 or an A in position 1.
IINPUT1    NS  01    1 CS
I         OR         1 CA
I                                  1   25  FLD1
 * Record identifying indicator 02 is set on if the record read
 * contains XYZA in positions 1 through 4.
I          NS  02    1 CX    2 CY    3 CZ
I         AND        4 CA
I                                  1   15  FLDA
I                                 16   20  FLDB
 * Record identifying indicator 95 is set on if any record read
 * does not meet the requirements for record identifying indicators
 * 01 or 02.
I          NS  95
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IRcdname+++....Ri........................................................
 *
 * For an externally described file, record identifying indicator 10
 * is set on if the ITMREC record is read and record identifying
 * indicator 20 is set on if the SLSREC or COMREC records are read.
IITMREC        10
ISLSREC        20
ICOMREC        20
```

Figure 14. Examples of Record Identifying Indicators

Control Level Indicators (L1-L9)

A control level indicator is defined by an entry in positions 63 and 64 of the input specifications, designating an input field as a control field. It can then be used to condition calculation and output operations. The valid control level indicator entries are L1 through L9.

A control level indicator designates an input field as a control field. When a control field is read, the data in the control field is compared with the data in the same control field from the previous record. If the data differs, a control break occurs, and the control level indicator assigned to the control field is set on. You can then use control level indicators to condition operations that are to be processed only when all records with the same information in the control field have been read. Because the indicators stay on for both total time and the first detail time, they can also be used to condition total printing (last record of a control group) or detail printing (first record in a control group). Control level indicators are set off before the next record is read.

A control break can occur after the first record containing a control field is read. The control fields in this record are compared to an area in storage that contains hexadecimal zeros. Because fields from two different records are not being compared, total calculations and total output operations are bypassed for this cycle.

Control level indicators are ranked in order of importance with L1 being the lowest and L9 the highest. All lower level indicators are set on when a higher level indicator is set on as the result of a control break. However, the lower level indicators can be used in the program only if they have been defined. For example, if L8 is set on by a control break, L1 through L7 are also set on. The LR (last record) indicator is set on when the input files are at end of file. LR is considered the highest level indicator and forces L1 through L9 to be set on.

You can also define control level indicators as record identifying or resulting indicators. When you use them in this manner, the status of the lower level indicators is not changed when a higher level indicator is set on. For example, if L3 is used as a resulting indicator, the status of L2 and L1 would not change if L3 is set on.

The importance of a control field in relation to other fields determines how you assign control level indicators. For example, data that demands a subtotal should have a lower control level indicator than data that needs a final total. A control field containing department numbers should have a higher control level indicator than a control field containing employee numbers if employees are to be grouped within departments (see Figure 15 on page 52).

Rules for Control Level Indicators

When you assign control level indicators, remember the following:

- You can specify control fields only for primary or secondary files.
- You cannot specify control fields for full procedural files; numeric input fields of type binary, integer, unsigned or float; or look-ahead fields.
- You cannot use control level indicators when an array name is specified in positions 49 through 62 of the input specifications; however, you can use control level indicators with an array element. Control level indicators are not allowed for null-capable fields.
- Control level compare operations are processed for records in the order in which they are found, regardless of the file from which they come.
- If you use the same control level indicator in different record types or in different files, the control fields associated with that control level indicator must be the same length (see Figure 15 on page 52) except for date, time, and timestamp fields which need only match in type (that is, they can be different formats).
- The control level indicator field length is the length of a control level indicator in a record. For example, if L1 has a field length of 10 bytes in a record, the control level indicator field length for L1 is 10 positions.

  The control level indicator field length for split control fields is the sum of the lengths of all fields associated with a control level indicator in a record. If L2 has a split control field consisting of 3 fields of length: 12 bytes, 2 bytes and 4 bytes; then the control level indicator field length for L2 is 18 positions.

If multiple records use the same control level indicator, then the control level indicator field length is the length of only one record, not the sum of all the lengths of the records.

Within a program, the sum of the control level indicator field lengths of all control level indicators cannot exceed 256 positions.

- Record positions in control fields assigned different control level indicators can overlap in the same record type (see Figure 16 on page 52). For record types that require control or match fields, the total length of the control or match field must be less than or equal to 256. For example, in Figure 16 on page 52, 15 positions have been assigned to control levels.
- Field names are ignored in control level operations. Therefore, fields from different record types that have been assigned the same control level indicator can have the same name.
- Control levels need not be written in any sequence. An L2 entry can appear before L1. All lower level indicators need not be assigned.
- If different record types in a file do not have the same number of control fields, unwanted control breaks can occur.

Figure 17 on page 53 shows an example of how to avoid unwanted control breaks.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
A* EMPLOYEE MASTER FILE -- EMPMSTL
A          R EMPREC                    PFILE(EMPMSTL)
A            EMPLNO         6
A            DEPT           3
A            DIVSON         1
A*
A*                (ADDITIONAL FIELDS)
A*
A          R EMPTIM                    PFILE(EMPMSTP)
A            EMPLNO         6
A            DEPT           3
A            DIVSON         1
A*
A*                (ADDITIONAL FIELDS)
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC...............................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
 *
 *  In this example, control level indicators are defined for three
 *  fields.  The names of the control fields (DIVSON, DEPT, EMPLNO)
 *  give an indication of their relative importance.
 *  The division (DIVSON) is the most important group.
 *  It is given the highest control level indicator used (L3).
 *  The department (DEPT) ranks below the division;
 *  L2 is assigned to it.  The employee field (EMPLNO) has
 *  the lowest control level indicator (L1) assigned to it.
 *
IEMPREC        10
I                                           EMPLNO        L1
I                                           DIVSON        L3
I                                           DEPT          L2
 *
 *  The same control level indicators can be used for different record
 *  types.  However, the control fields having the same indicators must
 *  be the same length.  For records in an externally described file,
 *  the field attributes are defined in the external description.
 *
IEMPTIM        20
I                                           EMPLNO        L1
I                                           DEPT          L2
I                                           DIVSON        L3
```

Figure 15. Control Level Indicators (Two Record Types)

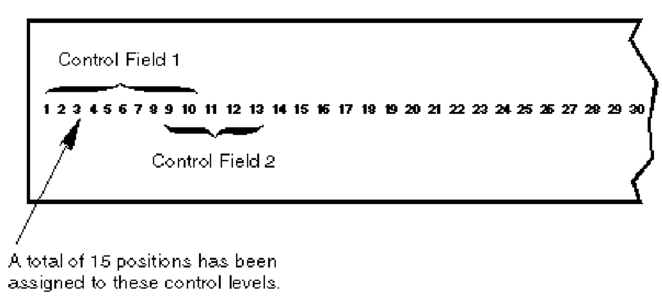


Figure 16. Overlapping Control Fields

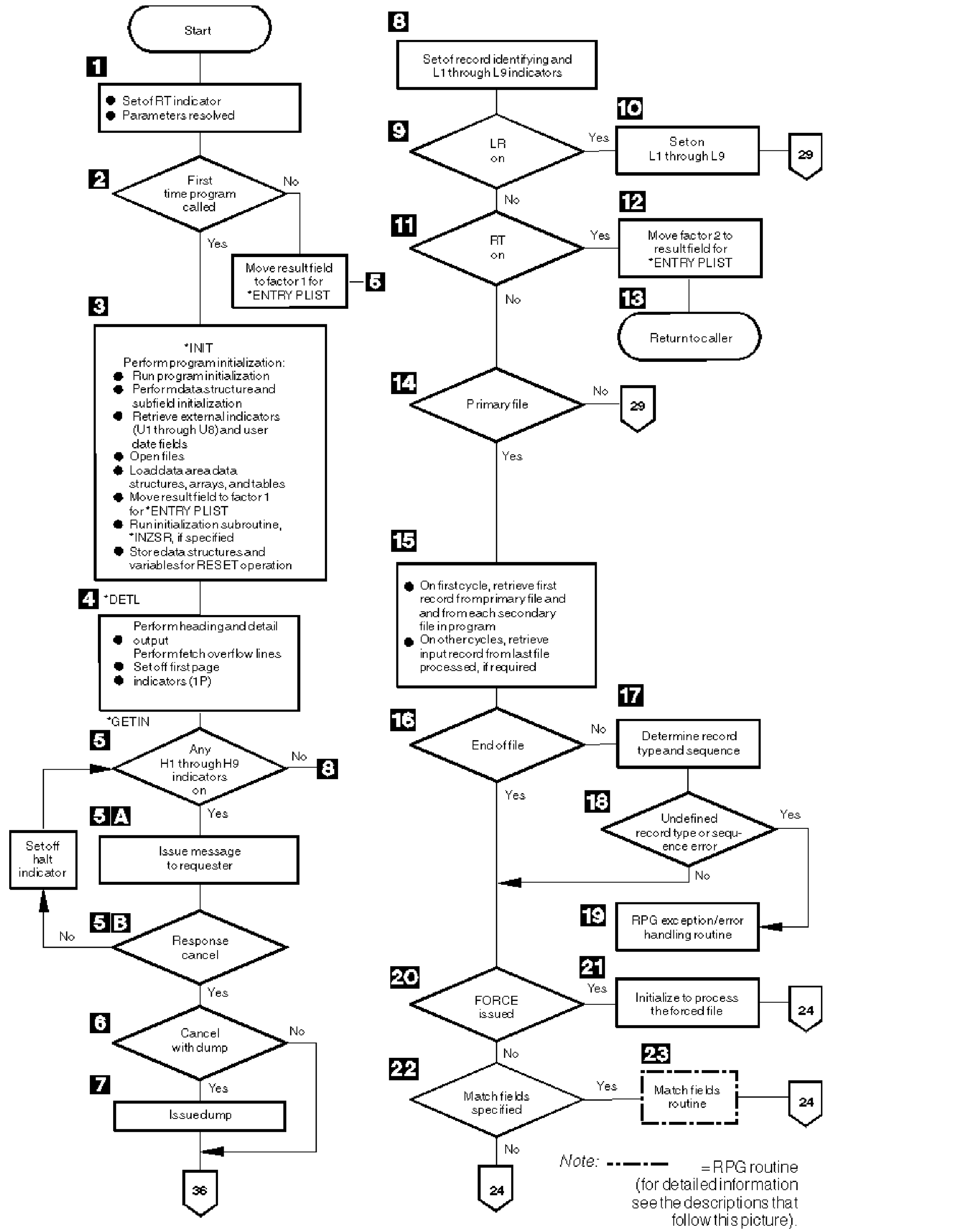


Figure 17. How to Avoid Unwanted Control Breaks (Part 1 of 4)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
ISALES         01
I                                  1    2  L2FLD         L2
I                                  3   15  NAME
IITEM          02
I                                  1    2  L2FLD         L2
I                                  3    5  L1FLD         L1
I                                  6    8  AMT
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *  Indicator 11 is set on when the salesman record is read.
 *
C   01              SETON                                        11
 *
 *  Indicator 11 is set off when the item record is read.
 *  This allows the normal L1 control break to occur.
 *
C   02              SETOFF                                       11
C   02AMT           ADD       L1TOT         L1TOT            5 0
CL1   L1TOT         ADD       L2TOT         L2TOT            5 0
CL2   L2TOT         ADD       LRTOT         LRTOT            5 0
 *
```

Figure 17. How to Avoid Unwanted Control Breaks (Part 2 of 4)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+...........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
OPRINTER   D    01                 1  1
O                       L2FLD             5
O                       NAME             25
O          D    02                    1
O                       L1FLD            15
O                       AMT           Z  15
 *
 *  When the next item record causes an L1 control break, no total
 *  output is printed if indicator 11 is on.  Detail calculations
 *  are then processed for the item record.
 *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+...........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
O          T    L1N11              1
O                     L1TOT           ZB  25
O                                         27 '*'
O          T    L2                 1
O                     L2TOT           ZB  25
O                                         28 '**'
O          T    LR                 1
O                     LRTOT           ZB  25
```

Figure 17. How to Avoid Unwanted Control Breaks (Part 3 of 4)

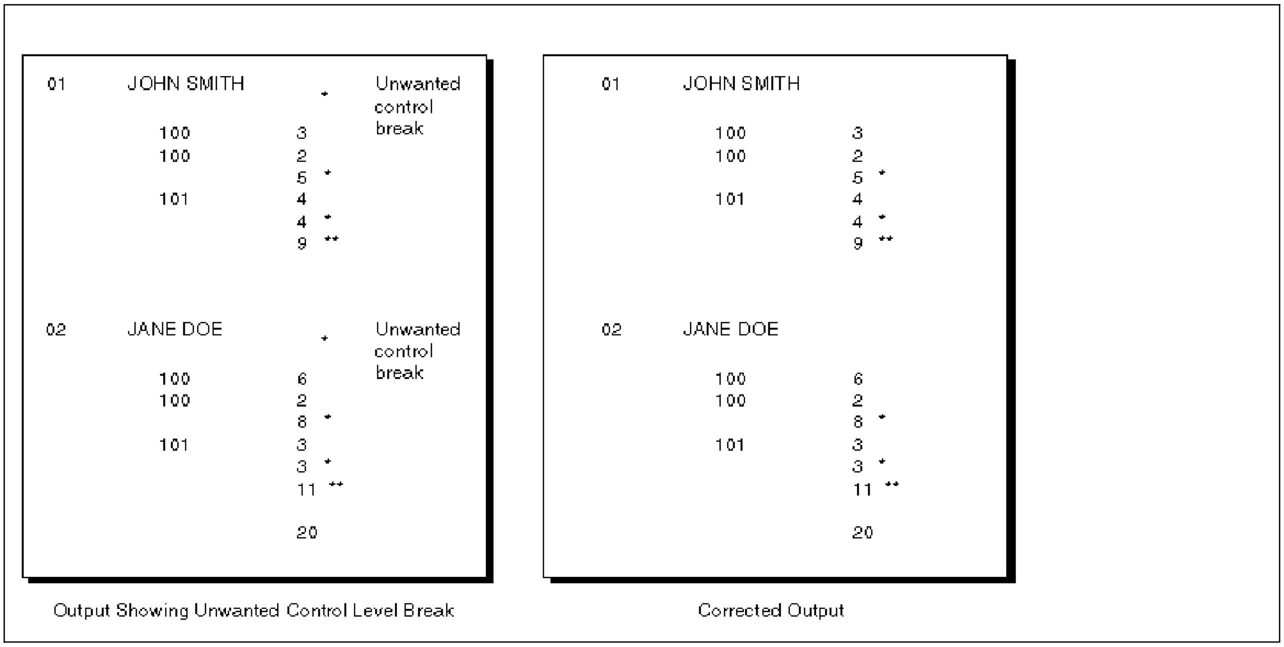


Figure 17. How to Avoid Unwanted Control Breaks (Part 4 of 4)

Different record types normally contain the same number of control fields. However, some applications require a different number of control fields in some records.

The salesman records contain only the L2 control field. The item records contain both L1 and L2 control fields. With normal RPG IV coding, an unwanted control break is created by the first item record following the salesman record. This is recognized by an L1 control break immediately following the salesman record and results in an asterisk being printed on the line below the salesman record.

- Numeric control fields are compared in zoned decimal format. Packed numeric input fields lengths can be determined by the formula:

  ```
  d = 2n - 1
  ```

  Where d = number of digits in the field and n = length of the input field. The number of digits in a packed numeric field is always odd; therefore, when a packed numeric field is compared with a zoned decimal numeric field, the zoned field must have an odd length.
- When numeric control fields with decimal positions are compared to determine whether a control break has occurred, they are always treated as if they had no decimal positions. For instance, 3.46 is considered equal to 346.
- If you specify a field as numeric, only the positive numeric value determines whether a control break has occurred; that is, a field is always considered to be positive. For example, -5 is considered equal to +5.
- Date and time fields are converted to *ISO format before being compared
- Graphic data is compared by hexadecimal value

Split Control Field

A split control field is formed when you assign more than one field in an input record the same control level indicator. For a program described file, the fields that have the same control level indicator are combined by the program in the order specified in the input specifications and treated as a single control field (see Figure 18 on page 56). The first field defined is placed in the high-order (leftmost) position of the control field, and the last field defined is placed in the low-order (rightmost) position of the control field.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IMASTER       01
I                                 28   31  CUSNO         L4
I                                 15   20  ACCTNO        L4
I                                 50   52  REGNO         L4
```

Figure 18. Split Control Fields

For an externally described file, fields that have the same control level indicator are combined in the order in which the fields are described in the data description specifications (DDS), not in the order in which the fields are specified on the input specifications. For example, if these fields are specified in DDS in the following order:

- EMPNO
- DPTNO
- REGNO

and if these fields are specified with the same control level indicator in the following order on the input specifications:

- REGNO L3
- DPTNO L3
- EMPNO L3

the fields are combined in the following order to form a split control field: EMPNO DPTNO REGNO.

Some special rules for split control fields are:

- For one control level indicator, you can split a field in some record types and not in others if the field names are different. However, the length of the field, whether split or not, must be the same in all record types.
- You can vary the length of the portions of a split control field for different record types if the field names are different. However, the total length of the portions must always be the same.
- A split control field can be made up of a combination of packed decimal fields and zoned decimal fields so long as the field lengths (in digits or characters) are the same.
- You must assign all portions of a split control field in one record type the same field record relation indicator and it must be defined on consecutive specification lines.
- When a split control field contains a date, time, or timestamp field than all fields in the split control field must be of the same type.

Figure 19 on page 57 shows examples of the preceding rules.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IDISK      BC  91   95 C1
I         OR   92   95 C2
I         OR   93   95 C3
I
 * All portions of the split control field must be assigned the same
 * control level indicator and all must have the same field record
 * relation entry.
I                                1    5  FLD1A         L1
I                               46   50  FLD1B         L1
I                               11   13  FLDA          L2
I                               51   60  FLD2A         L3
I                               31   40  FLD2B         L3
I                               71   75  FLD3A         L4  92
I                               26   27  FLD3B         L4  92
I                               41   45  FLD3C         L4  92
I                               61   70  FLDB              92
I                               21   25  FLDC              92
I                                6   10  FLD3D         L4  93
I                               14   20  FLD3E         L4  93
```

Figure 19. Split Control Fields–Special Rules

The record identified by a '1' in position 95 has two split control fields:

1. FLD1A and FLD1B
2. FLD2A and FLD2B

The record identified with a '2' in position 95 has three split control fields:

1. FLD1A and FLD1B
2. FLD2A and FLD2B
3. FLD3A, FLD3B, and FLD3C

The third record type, identified by the 3 in position 95, also has three split control fields:

1. FLD1A and FLD1B
2. FLD2A and FLD2B
3. FLD3D and FLD3E

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 valid field indicators are:

- 01-99
- H1-H9
- U1-U8
- RT

You can use a field indicator 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 only; 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; and 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. You can then use the field indicator 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.

You can use halt indicators (H1 through H9) as field indicators to check for an error condition in the field or array element as it is read into the program.

Rules for Assigning Field Indicators

When you assign field indicators, remember the following:

- 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 or with look-ahead fields. However, an entry can be made for an array element. Field indicators are allowed for null-capable fields only if ALWNULL(*USRCTL) is used.
- 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, especially when match fields or level control is involved.
- 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” on page 430.

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 22, “Operation Codes,” on page 607 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 valid resulting indicators are:
- 01-99
- H1-H9
- OA-OG, OV
- L1-L9
- LR
- U1-U8
- KA-KN, KP-KY (valid only with SETOFF)
- RT

You can specify resulting indicators 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, notably "LOOKUP (Look Up a Table or Array Element)" on page 711, "SETOFF (Set Indicator Off)" on page 812, and "SETON (Set Indicator On)" on page 813. A resulting indicator can be used as a conditioning indicator on the same calculation line or in other calculations or output operations. When you use it 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 20 on page 60).

Rules for Assigning Resulting Indicators

When assigning resulting indicators, remember the following:
- 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.
- When L1 through L9 indicators are used as resulting indicators and are set on, lower level indicators are not set on. For example, if L8 is set on, L1 through L7 are not set on.
- If H1 through H9 indicators are set on when used as resulting indicators, the program halts unless the halt indicator is set off prior to being checked in the program cycle. (See "RPG Cycle and other implicit Logic" on page 31).
- 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...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 *  Two resulting indicators are used to test for the different
 *  conditions in a subtraction operation.  These indicators are
 *  used to condition the calculations that must be processed for
 *  a payroll job.  Indicator 10 is set on if the hours worked (HRSWKD)
 *  are greater than 40 and is then used to condition all operations
 *  necessary to find overtime pay.  If Indicator 20 is not on
 *  (the employee worked 40 or more hours), regular pay based on a
 *  40-hour week is calculated.
 *
C     HRSWKD        SUB       40            OVERTM            3 01020
 *
C  N20PAYRAT        MULT (H)  40            PAY               6 2
C   10OVERTM        MULT (H)  OVRRAT        OVRPAY            6 2
C   10OVRPAY        ADD       PAY           PAY
 *
 *  If indicator 20 is on (employee worked less than 40 hours), pay
 *  based on less than a 40-hour week is calculated.
C   20PAYRAT        MULT (H)  HRSWKD        PAY
 *
```

Figure 20. Resulting Indicators Used to Condition Operations

Indicators Not Defined on the RPG IV Specifications

Not all indicators that can be used as conditioning indicators in an RPG IV program are defined on the specification forms. External indicators (U1 through U8) are defined by a CL command or by a previous RPG IV program. Internal indicators (1P, LR, MR, and RT) are defined by the RPG IV program cycle itself.

External Indicators

The external indicators are U1 through U8. These indicators can be set in a CL program or in an RPG IV program. In a CL program, they can be set by the SWS (switch-setting) parameter on the CL commands CHGJOB (Change Job) or CRTJOBD (Create Job Description). In an RPG IV program, they can be set as a resulting indicator or field indicator.

The status of the external indicators can be changed in the program by specifying them as resulting indicators on the calculation specifications or as field indicators on the input specifications. However, changing the status of the IBM i job switches with a CL program during processing of an RPG IV program has no effect on the copy of the external indicators used by the RPG IV program. Setting the external indicators on or off in the program has no effect on file operations. File operations function according to the status of the U1 through U8 indicators when the program is initialized. However, when a program ends normally with LR on, the external indicators are copied back into storage, and their status reflects their last status in the RPG IV program. The current status of the external indicators can then be used by other programs.

Note: When using “RETURN (Return to Caller)” on page 795 with the LR indicator off, you are specifying a return without an end and, as a result, no external indicators are updated.

Internal Indicators

Internal indicators include:

- First page indicator

- Last record indicator
- Matching record indicator
- Return Indicator.

First Page Indicator (1P)

The first page (1P) indicator is set on by the RPG IV program when the program starts running and is set off by the RPG IV program after detail time output. The first record will be processed after detail time output. The 1P indicator can be used to condition heading or detail records that are to be written at 1P time. Do not use the 1P indicator in any of the following ways:

- To condition output fields that require data from input records; this is because the input data will not be available.
- To condition total or exception output lines
- In an AND relationship with control level indicators
- As a resulting indicator
- When MAIN or NOMAIN is specified on a control specification

Last Record Indicator (LR)

In a program that contains a primary file, the last record indicator (LR) is set on after the last record from a primary/secondary file has been processed, or it can be set on by the programmer.

The LR indicator can be used to condition calculation and output operations that are to be done at the end of the program. When the LR indicator is set on, all other control level indicators (L1 through L9) are also set on. If any of the indicators L1 through L9 have not been defined as control level indicators, as record identifying indicators, as resulting indicators, or by *INxx, the indicators are set on when LR is set on, but they cannot be used in other specifications.

In a program that does not contain a primary file, you can set the LR indicator on as one method to end the program. (For more information on how to end a program without a primary file, see "RPG Cycle and other implicit Logic" on page 31.) To set the LR indicator on, you can specify the LR indicator as a record identifying indicator or a resulting indicator. If LR is set on during detail calculations, all other control level indicators are set on at the beginning of the next cycle. LR and the record identifying indicators are both on throughout the remainder of the detail cycle, but the record identifying indicators are set off before LR total time.

Matching Record Indicator (MR)

The matching record indicator (MR) is associated with the matching field entries M1 through M9. It can only be used in a program when Match Fields are defined in the primary and at least one secondary file.

The MR indicator is set on when all the matching fields in a record of a secondary file match all the matching fields of a record in the primary file. It remains on during the complete processing of primary and secondary records. It is set off when all total calculations, total output, and overflow for the records have been processed.

At detail time, MR always indicates the matching status of the record just selected for processing; at total time, it reflects the matching status of the previous record. If all primary file records match all secondary file records, the MR indicator is always on.

Use the MR indicator as a field record relation indicator, or as a conditioning indicator in the calculation specifications or output specifications to indicate operations that are to be processed only when records match. The MR indicator cannot be specified as a resulting indicator.

For more information on Match Fields and multi-file processing, see Chapter 6, "General File Considerations," on page 107.

Return Indicator (RT)

You can use the return indicator (RT) to indicate to the internal RPG IV logic that control should be returned to the calling program. The test to determine if RT is on is made after the test for the status of LR and before the next record is read. If RT is on, control returns to the calling program. RT is set off when the program is called again.

Because the status of the RT indicator is checked after the halt indicators (H1 through H9) and LR indicator are tested, the status of the halt indicators or the LR indicator takes precedence over the status of the RT indicator. If both a halt indicator and the RT indicator are on, the halt indicator takes precedence. If both the LR indicator and RT indicator are on, the program ends normally.

RT can be set on as a record identifying indicator, a resulting indicator, or a field indicator. It can then be used as a conditioning indicator for calculation or output operations.

For a description of how RT can be used to return control to the calling program, see the chapter on calling programs in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Using Indicators

Indicators that you have defined as overflow indicators, control level indicators, record identifying indicators, field indicators, resulting indicators, *IN, *IN(xx), *INxx, or those that are defined by the RPG IV language 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.

Note: Indicators that control the cycle function solely as conditioning indicators when used in a MAIN or NOMAIN module; or in a subprocedure that is active, but where the cycle-main procedure of the module is not. Indicators that control the cycle include: LR, RT, H1-H9, and control level indicators.

File Conditioning

The file conditioning indicators are specified by the EXTIND keyword on the file description specifications. Only the external indicators U1 through U8 are valid for file conditioning. (The USROPN keyword can be used to specify that no implicit OPEN should be done.)

If the external indicator specified is off when the program is called, the file is not opened and no data transfer to or from the file will occur when the program is running. Primary and secondary input files are processed as if they were at end-of-file. The end-of-file indicator is set on for all READ operations to that file. Input, calculation, and output specifications for the file need not be conditioned by the external indicator.

Rules for File Conditioning

When you condition files, remember the following:

- A file conditioning entry can be made for input, output, update, or combined files.
- A file conditioning entry cannot be made for table or array input.
- Output files for tables can be conditioned by U1 through U8. If the indicator is off, the table is not written.
- A record address file can be conditioned by U1 through U8, but the file processed by the record address file cannot be conditioned by U1 through U8.
- If the indicator conditioning a primary file with matching records is off, the MR indicator is not set on.
- Input does not occur for an input, an update, or a combined file if the indicator conditioning the file is off. Any indicators defined on the associated Input specifications in positions 63-74 will be processed as usual using the existing values in the input fields.
- Data transfer to the file does not occur for an output, an update, or a combined file if the indicator conditioning the file is off. Any conditioning indicators, numeric editing, or blank after that are defined on the output specifications for these files will be processed as usual.
- If the indicator conditioning an input, an update, or a combined file is off, the file is considered to be at end of file. All defined resulting indicators are set off at the beginning of each specified I/O operation. The end-of-file indicator is set on for READ, READC, READE, READPE, and READP operations. CHAIN, EXFMT, SETGT, SETLL, and UNLOCK operations are ignored and all defined resulting indicators remain set off.

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
- H1-H9
- MR
- RT
- L1-L9
- U1-U8

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

You use field record relation indicators to 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

You can use a record identifying indicator (01 through 99) 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 21 on page 65).

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.

Control fields, which you define with an L1 through L9 indicator in positions 63 and 64 of the input specifications, and match fields, which are specified by a match value (M1 through M9) in positions 65 and 66 of the input specifications, can also be related to a particular record type in an OR relationship if a field record relation indicator is specified. Control fields or match fields in the OR relationship that do not have a field record relation indicator are used with all record types in the OR relationship.

If two control fields have the same control level indicator or two match fields have the same matching level value, a field record relation indicator can be assigned to just one of the match fields. In this case, only the field with the field record relation indicator is used when that indicator is on. If none of the field record relation indicators are on for that control field or match field, the field without a field record relation indicator is used. Control fields and match fields can only have entries of 01 through 99 or H1 through H9 in positions 67 and 68.

You can use positions 67 and 68 to specify that the program accepts and uses data from a particular field only when a certain condition occurs (for example, when records match, when a control break occurs, or when an external indicator is on). You can indicate the conditions under which the program accepts data from a field by specifying indicators L1 through L9, MR, or U1 through U8 in positions 67 and 68. Data from the field named in positions 49 through 62 is accepted only when the field record relation indicator is on.

External indicators are primarily used when file conditioning is specified with the "EXTIND(*INUx)" on page 296 keyword on the file description specifications. However, they can be used even though file conditioning is not specified.

A halt indicator (H1 through H9) in positions 67 and 68 relates a field to a record that is in an OR relationship and also has a halt indicator specified in positions 21 and 22.

Remember the following points when you use field record relation indicators:

- Control level (positions 63 and 64) and matching fields (positions 65 and 66) with the same field record relation indicator must be grouped together.
- Fields used for control level (positions 63 and 64) and matching field entries (positions 65 and 66) without a field record relation indicator must appear before those used with a field record relation indicator.
- Control level (positions 63 and 64) and matching fields (positions 65 and 66) with a field record relation indicator (positions 67 and 68) take precedence, when the indicator is on, over control level and matching fields of the same level without an indicator.
- Field record relations (positions 67 and 68) for matching and control level fields (positions 63 through 66) must be specified with record identifying indicators (01 through 99 or H1 through H9) from the main specification line or an OR relation line to which the matching field refers. If multiple record types are specified in

an OR relationship, an indicator that specifies the field relation can be used to relate matching and control level fields to the pertinent record type.

- Noncontrol level (positions 63 and 64) and matching field (positions 65 and 66) specifications can be interspersed with groups of field record relation entries (positions 67 and 68).
- The MR indicator can be used as a field record relation indicator to reduce processing time when certain fields of an input record are required only when a matching condition exists.
- The number of control levels (L1 through L9) specified for different record types in the OR relationship can differ. There can be no control level for certain record types and a number of control levels for other record types.
- If all matching fields (positions 65 and 66) are specified with field record relation indicators (positions 67 and 68), each field record relation indicator must have a complete set of matching fields associated with it.
- If one matching field is specified without a field record relation indicator, a complete set of matching fields must be specified for the fields without a field record relation indicator.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IREPORT    AA  14    1 C5
I        OR    16    1 C6
I                                 20   30  FLDB
I                                  2   10  FLDA             07
 *
 *  Indicator 07 was specified elsewhere in the program.
 *
I                                 40   50  FLDC             14
I                                 60   70  FLDD             16
```

Figure 21. Field Record Relation

The file 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.

Function Key Indicators

You can use function key indicators in a program that contains a WORKSTN device if the associated function keys are specified in data description specifications (DDS). Function keys are specified in DDS with the CFxx or CAxx keyword. For an example of using function key indicators with a WORKSTN file, see the WORKSTN chapter in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Function Key Indicator	Corresponding Function Key	Function Key Indicator	Corresponding Function Key
KA	1	KM	13

Function Key Indicator	Corresponding Function Key	Function Key Indicator	Corresponding Function Key
KB	2	KN	14
KC	3	KP	15
KD	4	KQ	16
KE	5	KR	17
KF	6	KS	18
KG	7	KT	19
KH	8	KU	20
KI	9	KV	21
KJ	10	KW	22
KK	11	KX	23
KL	12	KY	24

The function key indicators correspond to function keys 1 through 24. Function key indicator KA corresponds to function key 1, KB to function key 2 ... KY to function key 24.

Function key indicators that are set on can then be used to condition calculation or output operations. Function key indicators can be set off by the SETOFF operation.

Halt Indicators (H1-H9)

You can use the halt indicators (H1 through H9) to indicate errors that occur during the running of a program. The halt indicators can be set on as record identifying indicators, field indicators, or resulting indicators.

The halt indicators are tested at the *GETIN step of the RPG IV cycle (see "RPG Cycle and other implicit Logic" on page 31). If a halt indicator is on, a message is issued to the user. The following responses are valid:

- Set off the halt indicator and continue the program.
- Issue a dump and end the program.
- End the program with no dump.

If a halt indicator is on when a RETURN operation inside a cycle-main procedure is processed, or when the LR indicator is on, the called program ends abnormally. The calling program is informed that the called program ended with a halt indicator on.

Note: If the keyword MAIN or NOMAIN is specified on a control specification, then any halt indicators are ignored except as conditioning indicators.

For a detailed description of the steps that occur when a halt indicator is on, see the detailed flowchart of the RPG IV cycle in "RPG Cycle and other implicit Logic" on page 31.

Indicators Conditioning Calculations

Calculation specifications in the traditional format (C specifications) can include conditioning indicators in positions 7 and 8, and 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

You can specify control level indicators (L1 through L9 and LR) in positions 7 and 8 of the calculation specifications.

If positions 7 and 8 are blank, the calculation is processed at detail time, is a statement within a subroutine, or is a declarative statement. If indicators L1 through L9 are specified, the calculation is processed at total time only when the specified indicator is on. If the LR indicator is specified, the calculation is processed during the last total time.

Note: An L0 entry can be used to indicate that the calculation is a total calculation that is to be processed on every program cycle.

Positions 9-11

You can use positions 9 through 11 of the calculation specifications to specify indicators that control the conditions under which an operation is processed. You can specify N is position 9 to indicate that the indicator should be tested for the value of off ('0') The valid entries for positions 10 through 11 are:

- 01-99
- H1-H9
- MR
- OA-OG, OV
- L1-L9
- LR
- U1-U8
- KA-KN, KP-KY
- RT

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

- Overflow indicators (file description specifications "OFLIND(indicator)" on page 303
- Record identifying indicators (input specifications, positions 21 and 22)
- Control level indicators (input specifications, positions 63 and 64)
- Field indicators (input specifications, positions 69 through 74)
- Resulting indicators (calculation specifications, positions 71 through 76)
- External indicators
- Indicators are set on, such as LR and MR
- *IN array, *IN(xx) array element, or *INxx field (see "Indicators Referred to As Data" on page 73 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 positions 9. The indicators in grouped AND/OR lines, plus the control level indicators (if specified in positions 7 and 8), must all be exactly as specified before the operation is done as in Figure 22 on page 68.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
C   25
CAN L1             SUB       TOTAL         TOTAL         A
CL2 10
CANNL3TOTAL        MULT      05            SLSTAX        B
 *
```

Figure 22. Conditioning Operations (Control Level Indicators)

Assume that indicator 25 represents a record type and that a control level 2 break occurred when record type 25 was read. L1 and L2 are both on. All operations conditioned by the control level indicators in positions 7 and 8 are done before operations conditioned by control level indicators in positions 9 through 11. Therefore, the operation in **B** occurs before the operation in **A**. The operation in **A** is done on the first record of the new control group indicated by 25, whereas the operation in **B** is a total operation done for all records of the previous control group.

The operation in **B** can be done when the L2 indicator is on provided the other conditions are met: Indicator 10 must be on; the L3 indicator must not be on.

The operation conditioned by both L2 and NL3 is done only when a control level 2 break occurs. These two indicators are used together because this operation is not to be done when a control level 3 break occurs, even though L2 is also on.

Some special considerations you should know when using conditioning indicators in positions 9 through 11 are as follows:

- With externally described work station files, the conditioning indicators on the calculation specifications must be either defined in the RPG program or be defined in the DDS source for the workstation file.
- With program described workstation files, the indicators used for the workstation file are unknown at compile time of the RPG program. Thus indicators 01-99 are assumed to be declared and they can be used to condition the calculation specifications without defining them.
- Halt indicators can be used to end the program or to prevent the operation from being processed when a specified error condition is found in the input data or in another calculation. Using a halt indicator is necessary because the record that causes the halt is completely processed before the program stops. Therefore, if the operation is processed on an error condition, the results are in error. A halt indicator can also be used to condition an operation that is to be done only when an error occurs.
- If LR is specified in positions 9 through 11, the calculation is done after the last record has been processed or after LR is set on.
- If a control level indicator is used in positions 9 through 11 and positions 7 and 8 are not used (detail time), the operation conditioned by the indicator is done only on the record that causes a control break or any higher level control break.
- If a control level indicator is specified in positions 7 and 8 (total time) and MR is specified in positions 9 through 11, MR indicates the matching condition of the previous record and not the one just read that caused the control break. After all operations conditioned by control level indicators in positions 7 and 8 are done, MR then indicates the matching condition of the record just read.
- If positions 7 and 8 and positions 9 through 11 are blank, the calculation specified on the line is done at detail calculation time.

Figure 23 and Figure 24 show examples of conditioning indicators.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilenameSqNORiPos1NCCPos2NCCPos3NCC.PFromTo++DField+L1M1FrPlMnZr...*
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
 *
 *  Field indicators can be used to condition operations.  Assume the
 *  program is to find weekly earnings including overtime.  The over-
 *  time field is checked to determine if overtime was entered.
 *  If the employee has worked overtime, the field is positive and -
 *  indicator 10 is set on.  In all cases the weekly regular wage
 *  is calculated.  However, overtime pay is added only if
 *  indicator 10 is on.
 *
ITIME      AB  01
I                                  1    7  EMPLNO
I                                  8   10 0OVERTM             10
I                                 15   20 2RATE
I                                 21   25 2RATEOT
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
 *
 *  Field indicator 10 was assigned on the input specifications.
 *  It is used here to condition calculation operations.
 *
C                   EVAL (H)  PAY = RATE * 40
C   10              EVAL (H)  PAY = PAY + (OVERTM * RATEOT)
```

Figure 23. Conditioning Operations (Field Indicators)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
 *
 *  A record identifying indicator is used to condition an operation.
 *  When a record is read with a T in position 1, the 01 indicator is
 *  set on.  If this indicator is on, the field named SAVE is added
 *  to SUM.  When a record without T in position 1 is read, the 02
 *  indicator is set on.  The subtract operation, conditioned by 02,
 *  then performed instead of the add operation.
 *
IFILE      AA  01    1 CT
I         OR   02    1NCT
I                                 10   15 2SAVE
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 *  Record identifying indicators 01 and 02 are assigned on the input
 *  specifications.  They are used here to condition calculation
 *  operations.
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C   01              ADD       SAVE          SUM                8 2
C   02              SUB       SAVE          SUM                8 2
```

Figure 24. 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). The following examples demonstrate this.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
 * In these examples, the IF structure is performed only if 01 is on.
 * *IN01 is treated as a boolean with a value of on or off.

 * In the first example, the value of the indicator ('0' or '1') is
 * checked.
C                   IF        *IN01

 * In the second example, the logical expression B < A is evaluated.
 * If true, 01 is set on.  If false 01 is set off.  This is analogous
 * to using COMP with A and B and placing 01 in the appropriate
 * resulting indicator position.
C                   EVAL      *IN01 = B < A
```

Figure 25. Indicators Used in Expressions

See the expressions chapter and the operation codes chapter in this document for more examples and further details.

Indicators Conditioning Output

Indicators that you use 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:

- Overflow indicators (file description specifications, "OFLIND(indicator)" on page 303)
- Record identifying indicators (input specifications, positions 21 and 22)
- Control level indicators (input specifications, positions 63 and 64)
- Field indicators (input specifications, positions 69 through 74)
- Resulting indicators (calculation specifications, positions 71 through 76)
- Indicators set by the RPG IV program such as 1P and LR
- External indicators set prior to or during program processing
- *IN array, *IN(xx) array element, or *INxx field (see "Indicators Referred to As Data" on page 73 for a description of how an indicator is defined when used with one of these reserved words).

If an indicator is to condition an entire record, you enter the indicator on the line that specifies the record type (see Figure 26 on page 72). If an indicator is to condition when a field is to be written, you enter the indicator on the same line as the field name (see Figure 26 on page 72).

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 you specify conditioning indicators, 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. If all negative indicators condition a heading or detail operation, the operation is done at the beginning of the program cycle when the first page (1P) lines are written.

You can specify output indicators 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, you can condition a field with more than three indicators by using the EVAL operation in calculations. The following example illustrates this.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
 * Indicator 20 is set on only if indicators 10, 12, 14,16, and 18
 * are set on.
C                   EVAL      *IN20 = *IN10 AND *IN12 AND *IN14
C                              AND *IN16 AND *IN18
OFilename++DAddN01N02N03Excnam++++.......................................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
 * OUTFIELD is conditioned by indicator 20, which effectively
 * means it is conditioned by all the indicators in the EVAL
 * operation.
OPRINTER   E
O               20      OUTFIELD
```

Other special considerations you should know about for output indicators are as follows:

- The first page indicator (1P) allows output on the first cycle before the primary file read, such as printing on the first page. The line conditioned by the 1P indicator must contain constant information used as headings or fields for reserved words such as PAGE and UDATE. The constant information is specified in the output specifications in positions 53 through 80. If 1P is used in an OR relationship with an overflow indicator, the information is printed on every page (see Figure 27 on page 72). Use the 1P indicator only with heading or detail output lines. It cannot be used to condition total or exception output lines or should not be used in an AND relationship with control level indicators.
- If certain error conditions occur, you might not want output operation processed. Use halt indicators to prevent the data that caused the error from being used (see Figure 28 on page 73).
- To condition certain output records on external conditions, use external indicators to condition those records.

See the Printer File section in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for a discussion of the considerations that apply to assigning overflow indicators on the output specifications.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+..........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
 *
 *  One indicator is used to condition an entire line of printing.
 *  When 44 is on, the fields named INVOIC, AMOUNT, CUSTR, and SALSMN
 *  are all printed.
 *
OPRINT     D    44                    1
O                       INVOIC             10
O                       AMOUNT             18
O                       CUSTR              65
O                       SALSMN             85
 *
 *  A control level indicator is used to condition when a field should
 *  be printed.  When indicator 44 is on, fields INVOIC, AMOUNT, and
 *  CUSTR are always printed.  However, SALSMN is printed for the
 *  first record of a new control group only if 44 and L1 are on.
 *
OPRINT     D    44                 1
O                       INVOIC             10
O                       AMOUNT             18
O                       CUSTR              65
O                  L1   SALSMN             85
```

Figure 26. Output Indicators

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+..........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
 *
 *  The 1P indicator is used when headings are to be printed
 *  on the first page only.
 *
OPRINT     H      1P                  3
O                                          8 'ACCOUNT'
 *
 *  The 1P indicator and an overflow indicator can be used to print
 *  headings on every page.
 *
OPRINT     H    1P                    3  1
O         OR    OF
O                                          8 'ACCOUNT'
```

Figure 27. 1P Indicator

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
 *
 *  When an error condition (zero in FIELDB) is found, the halt
 *  indicator is set on.
 *
IDISK      AA  01
I                                  1   3  FIELDA        L1
I                                  4   8 0FIELDB                  H1
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 *  When H1 is on, all calculations are bypassed.
 *
C   H1              GOTO      END
C                     :
C                     :                Calculations
C                     :
C     END           TAG
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+...........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat
 *
 *  FIELDA and FIELDB are printed only if H1 is not on.
 *  Use this general format when you do not want information that
 *  is in error to be printed.
 *
OPRINT     H    L1                 0  2 01
O                                          50 'HEADING'
O          D    01NH1                 1  0
O                       FIELDA              5
O                       FIELDB        Z     15
```

Figure 28. Preventing Fields from Printing

Indicators Referred to As Data

An alternative method of referring to and manipulating RPG IV indicators is provided by the RPG IV reserved words *IN and *INxx.

*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 RPG IV indicators except 1P or MR.

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.

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

Additional Rules

Remember the following rules when you are working with the array *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. Any subsequent normal RPG IV indicator tests may yield unpredictable results.
- 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 or *INL1, only that indicator will be defined.

See Figure 29 for some examples of indicators referred to as data.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 *  When this program is called, a single parameter is passed to
 *  control some logic in the program.  The parameter sets the value
 *  of indicator 50.  The parameter must be passed with a character
 *  value of 1 or 0.
 *
C     *ENTRY        PLIST
C     *IN50         PARM                    SWITCH            1
 *
 *
 *  Subroutine SUB1 uses indicators 61 through 68.  Before the
 *  subroutine is processed, the status of these indicators used in
 *  the mainline program is saved.  (Assume that the indicators are
 *  set off in the beginning of the subroutine.)  After the subroutine
 *  is processed, the indicators are returned to their original state.
 *
 *
C                   MOVEA     *IN(61)        SAV8             8
C                   EXSR      SUB1
C                   MOVEA     SAV8          *IN(61)
 *
 *  A code field (CODE) contains a numeric value of 1 to 5 and is
 *  used to set indicators 71 through 75.  The five indicators are set
 *  off.  Field X is calculated as 70 plus the CODE field.  Field X is
 *  then used as the index into the array *IN.  Different subroutines
 *  are then used based on the status of indicators 71 through 75.
 *
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 29. Examples of Indicators Referred to as Data

Summary of Indicators

Table 22 and Table 23 on page 76 show summaries of where RPG IV indicators are defined, what the valid entries are, where the indicators are used, and when the indicators are set on and off. Table 23 indicates the primary condition that causes each type of indicator to be set on and set off by the RPG IV program. "Function Key Indicators" on page 65 lists the function key indicators and the corresponding function keys.

Table 22. Indicator Entries and Uses

	Where Defined/Used	**01-99**	**1P**	**H1-H9**	**L1-L9**	**LR**	**MR**	**OA-OG OV**	**U1-U8**	**KA-KN KP-KY**	**RT**
User Defined	Overflow indicator, file description specifications, OFLIND keyword	X						X			
	Record identifying indicator input specifications, positions 21-22	X		X	X	X			X		X
	Control level, input specifications, positions 63-64				X						
	Field level, input specifications, positions 69-74	X		X					X		X
	Resulting indicator, calculation specifications, positions 71-76	X		X	X	X		X[1]	X	X[2]	X
RPG Defined	Internal Indicator		X			X	X				X
	External Indicator								X		
Used	File conditioning, file description specifications								X		
	File record relation, input specifications 67-68[3]	X		X	X		X		X		X
	Control level, calculation specifications, positions 7-8				X	X					
	Conditioning indicators, calculation specifications, positions 9-11	X		X	X	X	X	X	X	X	X
	Output indicators, output specifications, positions 21-29	X	X[4]	X	X	X	X	X	X	X	X

Table 22. Indicator Entries and Uses (continued)

	Where Defined/Used	01-99	1P	H1-H9	L1-L9	LR	MR	OA-OG OV	U1-U8	KA-KN KP-KY	RT

Notes:

1. The overflow indicator must be defined on the file description specification first.
2. KA through KN and KP through KY can be used as resulting indicators only with the SETOFF operation.
3. Only a record identifying indicator from a main or OR record can be used to condition a control or match field. L1 or L9 cannot be used to condition a control or match field.
4. The 1P indicator is allowed only on heading and detail lines.

Table 23. When Indicators Are Set On and Off by the RPG IV Logic Cycle

Type of Indicator	Set On	Set Off
Overflow	When printing on or spacing or skipping past the overflow line.	OA-OG, OV: After the following heading and detail lines are completed, or after the file is opened unless the H-specification keyword OPENOPT(*NOINZOFL) is used. 01-99: By the user.
Record identifying	When specified primary / secondary record has been read and before total calculations are processed; immediately after record is read from a full procedural file.	Before the next primary/secondary record is read during the next processing cycle.
Control level	When the value in a control field changes. All lower level indicators are also set on.	At end of following detail cycle.
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.
Function key	When the corresponding function key is pressed for WORKSTN files and at subsequent reads to associated subfiles.	By SETOFF or move fields logic for a WORKSTN file.
External U1-U8	By CL command prior to beginning the program, or when used as a resulting or a field indicator. **Note:** The value of the external indicators is set from the job switches during initialization. For a cycle module, it is done during the *INIT phase of the cycle; for other modules, it is done only once, when the first procedure in the module is called.	By CL command prior to beginning the program, or when used as a resulting or when used as a resulting or a field indicator.
H1-H9	As specified by programmer.	When the continue option is selected as a response to a message, or by the programmer.
RT	As specified by programmer.	When the program is called again.
Internal Indicators 1P	At beginning of processing before any input records are read.	Before the first record is read.
LR	After processing the last primary/secondary record of the last file or by the programmer.	At the beginning of processing, or by the programmer.

Table 23. When Indicators Are Set On and Off by the RPG IV Logic Cycle (continued)

Type of Indicator	Set On	Set Off
MR	If the match field contents of the record of a secondary file correspond to the match field contents of a record in the primary file.	When all total calculations and output are completed for the last record of the matching group.

Chapter 5. File and Program Exception/Errors

RPG categorizes exception/errors into two classes: program and file. Information on file and program exception/errors is made available to an RPG IV 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.

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.
- You can create a user-defined ILE exception handler that will take control when an exception occurs. For more information, see *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.
- A file exception/error subroutine can be specified for a global file in a cycle module. 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 file exception/error subroutine is not present, any file exception/errors are handled by the RPG IV default error handler.

File Information Data Structure

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 or procedure.

The file information data structure, which must be unique for each file, must be defined in the same scope as the file. For global files, the INFDS must be defined in the main source section. For local files in a subprocedure, the INFDS must be defined in the Definition specifications of the subprocedure. Furthermore, the INFDS must be defined with the same storage type, automatic or static, as the file.

The INFDS for a file is used by all procedures using the file. If the file is passed as a parameter, the called program or procedure uses the same INFDS.

The INFDS contains the following feedback information:
- File Feedback (length is 80)
- Open Feedback (length is 160)
- Input/Output Feedback (length is 126)
- Device Specific Feedback (length is variable)
- Get Attributes Feedback (length is variable)

Note: The get attributes feedback uses the same positions in the INFDS as the input/output feedback and device specific feedback. This means that if you have a get attributes feedback, you cannot have input/output feedback or device feedback, and vice versa.

The length of the INFDS depends on what fields you have declared in your INFDS. The minimum length of the INFDS is 80.

File Feedback Information

The file feedback information starts in position 1 and ends in position 80 in the file information data structure. The file feedback information contains data about the file which is specific to RPG. This includes information about the error/exception that identifies:
- The name of the file for which the exception/error occurred
- The record being processed when the exception/error occurred or the record that caused the exception/error
- The last operation being processed when the exception/error occurred
- The status code
- The RPG IV routine in which the exception/error occurred.

The fields from position 1 to position 66 in the file feedback section of the INFDS are always provided and updated even if INFDS is not specified in the program. The fields from position 67 to position 80 of the file feedback section of the INFDS are only updated after a POST operation to a specific device.

If INFDS is not specified, the information in the file feedback section of the INFDS can be output using the DUMP operation. For more information see “DUMP (Program Dump)” on page 669.

Overwriting the file feedback section of the INFDS may 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 of the INFDS is defined by special keywords. The contents of the file feedback section of the INFDS along with the special keywords and their descriptions can be found in the following tables:

Table 24. Contents of the File Feedback Information Available in the File Information Data Structure (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. For a description of these codes, see "File Status Codes" on page 91.
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. If the operation was an implicit operation (for example, a primary file read or update on the output specifications), the equivalent operation code is generated (such as READ or UPDAT) and placed in location *OPCODE. Operation codes which have 6 letter names will 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 name of the routine (including a subprocedure) in which the file operation was done.

Table 24. Contents of the File Feedback Information Available in the File Information Data Structure (INFDS) (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
30	37	Character	8		If OPTION(*NOSRCSTMT) is specified, this is the source listing line number of the file operation. If OPTION(*SRCSTMT) is specified, this is the source listing statement number of the file operation. The full statement number is included when it applies to the root source member. If the statement number is greater than 6 digits, that is, it includes a source ID other than zero, the first 2 positions of the 8-byte feedback area will have a "+ " indicating that the rest of the statement number is stored in positions 53-54.
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/error occurred.
46	52	Character	7		Machine or system message number.
53	66	Character	14		Unused.
77	78	Binary	2		Source Id matching the statement number from positions 30-37.

Table 25. Contents of the File Feedback Information Available in the File-Information Data Structure (INFDS) Valid after a POST

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
67	70	Zoned decimal	4,0	*SIZE	Screen size (product of the number of rows and the number of columns on the device screen).
71	72	Zoned decimal	2,0	*INP	The display's keyboard type. Set to 00 if the keyboard is alphanumeric or katakana. Set to 10 if the keyboard is ideographic.
73	74	Zoned decimal	2,0	*OUT	The display type. Set to 00 if the display is alphanumeric or katakana. Set to 10 if the display is ideographic. Set to 20 if the display is DBCS.
75	76	Zoned decimal	2,0	*MODE	Always set to 00.

INFDS File Feedback Example: To specify an INFDS which contains fields in the file feedback section, you can make 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++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    IF   E             DISK    INFDS(FILEFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DFILEFBK          DS
D FILE              *FILE                                                    * File name
D OPEN_IND               9      9N                                           * File open?
D EOF_IND               10     10N                                           * 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
D SCREEN            *SIZE                                                    * Screen size
D NLS_IN            *INP                                                     * NLS Input?
D NLS_OUT           *OUT                                                     * NLS Output?
D NLS_MODE          *MODE                                                    * NLS Mode?
```

Figure 30. 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 the file open feedback area are copied by RPG to the open feedback section of the INFDS 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.

A description of the contents of the open feedback area, and what file types the fields are valid for, can be found in the iSeries Information Center.

INFDS Open Feedback Example: To specify an INFDS which contains fields in the open feedback section, you can make 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 iSeries Information Center database and file systems category 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 of the INFDS, 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)
  ```

  For example, for overflow line number of a printer file, the Information Center gives:

  ```
  Offset = 107
  Data Type is binary
  Length = 2
  ```
  Therefore,

```
From = 81 + 107 = 188,
To = 188 - 1 + 2 = 189.
See subfield OVERFLOW in example below
```

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    O    F  132        PRINTER INFDS(OPNFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DOPNFBK          DS
D ODP_TYPE              81     82                                       * ODP Type
D FILE_NAME             83     92                                       * File name
D LIBRARY               93    102                                       * Library name
D SPOOL_FILE           103    112                                       * Spool file name
D SPOOL_LIB            113    122                                       * Spool file lib
D SPOOL_NUM_OLD        123    124I 0                                    * Spool file num
D RCD_LEN              125    126I 0                                    * Max record len
D KEY_LEN              127    128I 0                                    * Max key len
D MEMBER               129    138                                       * Member name
D TYPE                 147    148I 0                                    * File type
D ROWS                 152    153I 0                                    * Num PRT/DSP rows
D COLUMNS              154    155I 0                                    * Num PRT/DSP cols
D NUM_RCDS             156    159I 0                                    * Num of records
D SPOOL_NUM            160    163I 0                                    * 6 digit Spool Nbr
D VOL_OFF              184    185I 0                                    * Vol label offset
D BLK_RCDS             186    187I 0                                    * Max rcds in blk
D OVERFLOW             188    189I 0                                    * Overflow line
D BLK_INCR             190    191I 0                                    * Blk increment
D FLAGS1               196    196                                       * Misc flags
D REQUESTER            197    206                                       * Requester name
D OPEN_COUNT           207    208I 0                                    * Open count
D BASED_MBRS           211    212I 0                                    * Num based mbrs
D FLAGS2               213    213                                       * Misc flags
D OPEN_ID              214    215                                       * Open identifier
D RCDFMT_LEN           216    217I 0                                    * Max rcd fmt len
D CCSID                218    219I 0                                    * Database CCSID
D FLAGS3               220    220                                       * Misc flags
D NUM_DEVS             227    228I 0                                    * Num devs defined
```

Figure 31. Example of Coding an INFDS with Open Feedback Information

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 by RPG to the input/output feedback section of the INFDS:

- If the presence of a POST operation affects the file:
  - only after a POST for the file.
- Otherwise:
  - after each I/O operation, if blocking is not active for the file.
  - after the I/O request to data management to get or put a block of data, if blocking is active for the file.

For more information see "POST (Post)" on page 770.

A description of the contents of the input/output feedback area can be found in the Information Center.

INFDS Input/Output Feedback Example: To specify an INFDS which contains fields in the input/output feedback section, you can make 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 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 of the INFDS, use the Offset, Data Type, and Length given in the 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, the 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 example below
```

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    IF   E             DISK    INFDS(MYIOFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DMYIOFBK          DS
D                                                                          * 241-242 not used
D WRITE_CNT             243    246I 0                                      * Write count
D READ_CNT              247    250I 0                                      * Read count
D WRTRD_CNT             251    254I 0                                      * Write/read count
D OTHER_CNT             255    258I 0                                      * Other I/O count
D OPERATION             260    260                                         * Cuurent 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    286I 0                                      * Rcd len of I/O
```

Figure 32. Example of Coding an INFDS with 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, and contains input/output feedback information specific to a device.

The length of the INFDS when device specific feedback information is required, depends on two factors: the device type of the file, and on whether DISK files are keyed or not. The minimum length is 528; but some files require a longer INFDS.

- For WORKSTN files, the INFDS is long enough to hold the device-specific feedback information for any type of display or ICF file starting at position 241. For example, if the longest device-specific feedback information requires 390 bytes, the INFDS for WORKSTN files is 630 bytes long (240+390=630).
- For externally described DISK files, the INFDS is at least long enough to hold the longest key in the file beginning at position 401.

More information on the contents and length of the device feedback for database file, printer files, ICF and display files can be found in the iSeries Information Center database and file systems category.

The contents of the device specific input/output feedback area of the file are copied by RPG to the device specific feedback section of the INFDS:

- If the presence of a POST operation affects the file:
  - only after a POST for the file.
- Otherwise:
  - after each I/O operation, if blocking is not active for the file.
  - after the I/O request to data management to get or put a block of data, if blocking is active for the file.

Notes:

1. After each keyed input operation, only the key fields will be updated.
2. After each non-keyed input operation, only the relative record number will be updated.

For more information see “POST (Post)” on page 770.

INFDS Device Specific Feedback Examples: To specify an INFDS which contains fields in the device-specific feedback section, you can make 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 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 of the INFDS, 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 gives:

  ```
     Offset = 30
     Data Type is binary
     Length = 4
  Therefore,
     From = 367 + 30 = 397,
     To = 397 - 1 + 4 = 400.
     See subfield DB_RRN in DBFBK data structure in example below
  ```

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    O    F  132        PRINTER INFDS(PRTFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DPRTFBK           DS
D CUR_LINE              367    368I 0                                      * Current line num
D CUR_PAGE              369    372I 0                                      * Current page cnt
 * If the first bit of PRT_FLAGS is on, the spooled file has been
 * deleted.  Use TESTB X'80' or TESTB '0' to test this bit.
D PRT_FLAGS             373    373
D PRT_MAJOR             401    402                                         * Major ret code
D PRT_MINOR             403    404                                         * Minor ret code
```

Figure 33. Example of Coding an INFDS with Printer Specific Feedback Information

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    IF   E             DISK    INFDS(DBFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DDBFBK            DS
D FDBK_SIZE             367    370I 0                                      * Size of DB fdbk
D JOIN_BITS             371    374I 0                                      * JFILE bits
D LOCK_RCDS             377    378I 0                                      * Nbr locked rcds
D POS_BITS              385    385                                         * File pos bits
D DLT_BITS              384    384                                         * Rcd deleted bits
D NUM_KEYS              387    388I 0                                      * Num keys (bin)
D KEY_LEN               393    394I 0                                      * Key length
D MBR_NUM               395    396I 0                                      * Member number
D DB_RRN                397    400I 0                                      * Relative-rcd-num
D KEY                   401   2400                                         * Key value (max
D                                                                          *   size 2000)
```

Figure 34. Example of Coding an INFDS with Database Specific Feedback Information

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    CF   E             WORKSTN INFDS(ICFFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DICFFBK           DS
D ICF_AID               369    369                                         * AID byte
D ICF_LEN               372    375I 0                                      * Actual data len
D ICF_MAJOR             401    402                                         * Major ret code
D ICF_MINOR             403    404                                         * Minor ret code
D SNA_SENSE             405    412                                         * SNA sense rc
D SAFE_IND              413    413                                         * Safe indicator
D RQSWRT                415    415                                         * Request write
D RMT_FMT               416    425                                         * Remote rcd fmt
D ICF_MODE              430    437                                         * Mode name
```

Figure 35. Example of Coding an INFDS with ICF Specific Feedback Information

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    CF   E             WORKSTN INFDS(DSPFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DDSPFBK            DS
D DSP_FLAG1             367    368                                         * Display flags
D DSP_AID               369    369                                         * AID byte
D CURSOR                370    371                                         * Cursor location
D DATA_LEN              372    375I 0                                      * Actual data len
D SF_RRN                376    377I 0                                      * Subfile rrn
D MIN_RRN               378    379I 0                                      * Subfile min rrn
D NUM_RCDS              380    381I 0                                      * Subfile num rcds
D ACT_CURS              382    383                                         * Active window
D                                                                          *  cursor location
D DSP_MAJOR             401    402                                         * Major ret code
D DSP_MINOR             403    404                                         * Minor ret code
```

Figure 36. Example of Coding an INFDS with Display Specific Feedback Information

Get Attributes Feedback Information

The get attributes feedback information in the file information data structure starts at position 241 in the INFDS, and contains information about a display device or ICF session (a device associated with a WORKSTN file). The end position of the get attributes feedback information depends on the length of the data returned by a get attributes data management operation. The get attributes data management operation is performed when a POST with a program device specified for factor 1 is used.

More information about the contents and the length of the get attributes data can be found in the Information Center.

INFDS Get Attributes Feedback Example: To specify an INFDS which contains fields in the get attributes feedback section, you can make 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 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 get attributes feedback section of the INFDS, use the Offset, Data Type, and Length given in the Information Center and do the following calculations:

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

  For example, for device type of a file, the Information Center gives:

```
   Offset = 31
   Data Type is character
   Length = 6
Therefore,
   From = 241 + 31 = 272,
   To = 272 - 1 + 6 = 277.
   See subfield DEV_TYPE in example below
```

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    CF   E             WORKSTN INFDS(DSPATRFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DDSPATRFBK        DS
D PGM_DEV               241    250                                         * Program device
D DEV_DSC               251    260                                         * Dev description
D USER_ID               261    270                                         * User ID
D DEV_CLASS             271    271                                         * Device class
D DEV_TYPE              272    277                                         * Device type
D REQ_DEV               278    278                                         * Requester?
D ACQ_STAT              279    279                                         * Acquire status
D INV_STAT              280    280                                         * Invite status
D DATA_AVAIL            281    281                                         * Data available
D NUM_ROWS              282    283I 0                                      * Number of rows
D NUM_COLS              284    285I 0                                      * Number of cols
D BLINK                 286    286                                         * Allow blink?
D LINE_STAT             287    287                                         * Online/offline?
D DSP_LOC               288    288                                         * Display location
D DSP_TYPE              289    289                                         * Display type
D KBD_TYPE              290    290                                         * Keyboard type
D CTL_INFO              342    342                                         * Controller info
D COLOR_DSP             343    343                                         * Color capable?
D GRID_DSP              344    344                                         * Grid line dsp?
 * The following fields apply to ISDN.
D ISDN_LEN              385    386I 0                                      * Rmt number len
D ISDN_TYPE             387    388                                         * Rmt number type
D ISDN_PLAN             389    390                                         * Rmt number plan
D ISDN_NUM              391    430                                         * Rmt number
D ISDN_SLEN             435    436I 0                                      * Rmt sub-address
D                                                                          *   length
D ISDN_STYPE            437    438                                         * Rmt sub-address
D                                                                          *   type
D ISDN_SNUM             439    478                                         * Rmt sub-address
D ISDN_CON              480    480                                         * Connection
D ISDN_RLEN             481    482I 0                                      * Rmt address len
D ISDN_RNUM             483    514                                         * Rmt address
D ISDN_ELEN             519    520                                         * Extension len
D ISDN_ETYPE            521    521                                         * Extension type
D ISDN_ENUM             522    561                                         * Extension num
D ISDN_XTYPE            566    566                                         * X.25 call type
D
```

Figure 37. Example of Coding an INFDS with Display file Get Attributes Feedback Information

```
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++
FMYFILE    CF   E             WORKSTN INFDS(ICFATRFBK)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++Comments++++++++++
DICFATRFBK        DS
D PGM_DEV               241    250                                         * Program device
D DEV_DSC               251    260                                         * Dev description
D USER_ID               261    270                                         * User ID
D DEV_CLASS             271    271                                         * Device class
D DEV_TYPE              272    272                                         * Device type
D REQ_DEV               278    278                                         * Requester?
D ACQ_STAT              279    279                                         * Acquire status
D INV_STAT              280    280                                         * Invite status
D DATA_AVAIL            281    281                                         * Data available
D SES_STAT              291    291                                         * Session status
D SYNC_LVL              292    292                                         * Synch level
D CONV_TYPE             293    293                                         * Conversation typ
D RMT_LOC               294    301                                         * Remote location
D LCL_LU                302    309                                         * Local LU name
D LCL_NETID             310    317                                         * Local net ID
D RMT_LU                318    325                                         * Remote LU
D RMT_NETID             326    333                                         * Remote net ID
D APPC_MODE             334    341                                         * APPC Mode
D LU6_STATE             345    345                                         * LU6 conv state
D LU6_COR               346    353                                         * LU6 conv
D                                                                          *    correlator
 * The following fields apply to ISDN.
D ISDN_LEN              385    386I 0                                      * Rmt number len
D ISDN_TYPE             387    388                                         * Rmt number type
D ISDN_PLAN             389    390                                         * Rmt number plan
D ISDN_NUM              391    430                                         * Rmt number
D ISDN_SLEN             435    436I 0                                      * sub-addr len
D ISDN_STYPE            437    438                                         * sub-addr type
D ISDN_SNUM             439    478                                         * Rmt sub-address
D ISDN_CON              480    480                                         * Connection
D ISDN_RLEN             481    482I 0                                      * Rmt address len
D ISDN_RNUM             483    514                                         * Rmt address
D ISDN_ELEN             519    520                                         * Extension len
D ISDN_ETYPE            521    521                                         * Extension type
D ISDN_ENUM             522    561                                         * Extension num
D ISDN_XTYPE            566    566                                         * X.25 call type

 * The following information is available only when program was started
 * as result of a received program start request. (P_ stands for protected)
D TRAN_PGM              567    630                                         * Trans pgm name
D P_LUWIDLN             631    631                                         * LUWID fld len
D P_LUNAMELN            632    632                                         * LU-NAME len
D P_LUNAME              633    649                                         * LU-NAME
D P_LUWIDIN             650    655                                         * LUWID instance
D P_LUWIDSEQ            656    657I 0                                      * LUWID seq num

 * The following information is available only when a protected conversation
 * is started on a remote system.  (U_ stands for unprotected)
D U_LUWIDLN             658    658                                         * LUWID fld len
D U_LUNAMELN            659    659                                         * LU-NAME len
D U_LUNAME              660    676                                         * LU-NAME
D U_LUWIDIN             677    682                                         * LUWID instance
D U_LUWIDSEQ            683    684I 0                                      * LUWID seq num
```

Figure 38. Example of Coding an INFDS with ICF file Get Attributes Feedback Information

Blocking Considerations

The fields of the input/output specific feedback in the INFDS and in most cases the fields of the device specific feedback information section of the INFDS, are not updated for each operation to the file in which the records are blocked and unblocked. The feedback information is updated only when a block of records is transferred between an RPG program and the operating system. However, if you are doing blocked input on a data base file, the relative record number and the key value in the data base feedback section of the INFDS are updated:

- On every input/output operation, if the file is not affected by the presence of a POST operation in the program.
- Only after a POST for the file, if file is affected by a POST operation in the program.

See “POST (Post)” on page 770.

You can obtain valid updated feedback information by using the CL command OVRDBF (Override with Database File) with SEQONLY(*NO) specified. If you use a file override command, the ILE RPG compiler does not block or unblock the records in the file.

For more information on blocking and unblocking of records in RPG see *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

File Status Codes

Any code placed in the subfield location *STATUS that is greater than 99 is considered to be an exception/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'; 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 codes in the following tables are placed in the subfield location *STATUS for the file information data structure:

Table 26. Normal Codes

Code	**Device**[1]	**RC**[2]	**Condition**
00000			No exception/error.
00002	W	n/a	Function key used to end display.
00011	W,D,SQ	11xx	End of file on a read (input).
00012	W,D,SQ	n/a	No-record-found condition on a CHAIN, SETLL, and SETGT operations.
00013	W	n/a	Subfile is full on WRITE operation.
Note: [1]“Device” refers to the devices for which the condition applies. The following abbreviations are used: P = PRINTER; D = DISK; W = WORKSTN; SP = SPECIAL; SQ = Sequential. The major/minor return codes under column RC apply only to WORKSTN files. [2]The formula mmnn is used to described major/minor return codes: mm is the major and nn the minor.			

File Exception/Errors

Table 27. Exception/Error Codes

Code	Device[1]	RC[2]	Condition
01011	W,D,SQ	n/a	Undefined record type (input record does not match record identifying indicator).
01021	W,D,SQ	n/a	Tried to write a record that already exists (file being used has unique keys and key is duplicate, or attempted to write duplicate relative record number to a subfile).
01022	D	n/a	Referential constraint error detected on file member.
01023	D,SQ	n/a	Error in trigger program before file operation performed.
01024	D,SQ	n/a	Error in trigger program after file operation performed.
01031	W,D,SQ	n/a	Match field out of sequence.
01041	n/a	n/a	Array/table load sequence error.
01042	n/a	n/a	Array/table load sequence error. Alternate collating sequence used.
01051	n/a	n/a	Excess entries in array/table file.
# 01061	n/a	n/a	Error handling for an associated variable for a file parameter
01071	W,D,SQ	n/a	Numeric sequence error.
01121[4]	W	n/a	No indicator on the DDS keyword for Print key.
01122[4]	W	n/a	No indicator on the DDS keyword for Roll Up key.
01123[4]	W	n/a	No indicator on the DDS keyword for Roll Down key.
01124[4]	W	n/a	No indicator on the DDS keyword for Clear key.
01125[4]	W	n/a	No indicator on the DDS keyword for Help key.
01126[4]	W	n/a	No indicator on the DDS keyword for Home key.
01201	W	34xx	Record mismatch detected on input.
01211	all	n/a	I/O operation to a closed file.
01215	all	n/a	OPEN issued to a file already opened.
01216[3]	all	yes	Error on an implicit OPEN/CLOSE operation.
01217[3]	all	yes	Error on an explicit OPEN/CLOSE operation.
01218	D,SQ	n/a	Record already locked.
01221	D,SQ	n/a	Update operation attempted without a prior read.
01222	D,SQ	n/a	Record cannot be allocated due to referential constraint error
01231	SP	n/a	Error on SPECIAL file.
01235	P	n/a	Error in PRTCTL space or skip entries.
01241	D,SQ	n/a	Record number not found. (Record number specified in record address file is not present in file being processed.)
01251	W	80xx 81xx	Permanent I/O error occurred.
01255	W	82xx 83xx	Session or device error occurred. Recovery may be possible.
01261	W	n/a	Attempt to exceed maximum number of acquired devices.
01271	W	n/a	Attempt to acquire unavailable device
01281	W	n/a	Operation to unacquired device.
01282	W	0309	Job ending with controlled option.
01284	W	n/a	Unable to acquire second device for single device file
01285	W	0800	Attempt to acquire a device already acquired.

Table 27. Exception/Error Codes (continued)

Code	Device[1]	RC[2]	Condition
01286	W	n/a	Attempt to open shared file with SAVDS or IND options.
01287	W	n/a	Response indicators overlap IND indicators.
01299	W,D,SQ	yes	Other I/O error detected.
01331	W	0310	Wait time exceeded for READ from WORKSTN file.

Notes:

1. "Device" refers to the devices for which the condition applies. The following abbreviations are used: P = PRINTER; D = DISK; W = WORKSTN; SP = SPECIAL; SQ = Sequential. The major/minor return codes under column RC apply only to WORKSTN files.
2. The formula mmnn is used to described major/minor return codes: mm is the major and nn the minor.
3. Any errors that occur during an open or close operation will result in a *STATUS value of 1216 or 1217 regardless of the major/minor return code value.
4. See Figure 11 on page 42 for special handling.

The following table shows the major/minor return code to *STATUS value mapping for errors that occur to AS/400 programs using WORKSTN files only. See the Information Center for more information on major/minor return codes.

Major	Minor	*STATUS
00,02	all	00000
03	all (except 09,10)	00000
03	09	01282
03	10	01331
04	all	01299
08	all	01285[1]
11	all	00011
34	all	01201
80,81	all	01251
82,83	all	01255

Notes:

1. The return code field will not be updated for a *STATUS value of 1285, 1261, or 1281 because these conditions are detected before calling data management. To monitor for these errors, you must check for the *STATUS value and not for the corresponding major/minor return code value.

File Exception/Error Subroutine (INFSR)

To identify the user-written RPG IV subroutine that may receive control following file exception/errors, specify the INFSR keyword on the File Description specification with the name of the subroutine that receives control when exception/errors occur on this file. The subroutine name can be *PSSR, which indicates that the program exception/error subroutine is given control for the exception/errors on this file.

A file exception/error subroutine (INFSR) receives control when an exception/error occurs on an implicit (primary or secondary) file operation or on an explicit 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 RPG IV operations 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).

Note: The INFSR keyword cannot be specified if the keyword MAIN or NOMAIN keyword is specified on the Control specification, or if the file is to be accessed by a subprocedure. To handle errors for the file in your procedure, you can use the (E) extender to handle errors for an individual I/O operation, or you can use a MONITOR group to handle errors for several operations. The ON-ERROR section of your MONITOR group could call a subprocedure to handle the details of the error handling.

The ENDSR operation must be the last specification for the file exception/error subroutine and should 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 6-position 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.

***DETL**
Continue at the beginning of detail lines.

***GETIN**
Continue at the get input record routine.

***TOTC**
Continue at the beginning of total calculations.

***TOTL**
Continue at the beginning of total lines.

***OFL** Continue at the beginning of overflow lines.

***DETC**
Continue at the beginning of detail calculations.

***CANCL**
Cancel the processing of the program.

Blanks Return control to the RPG IV 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 runtime.

50-76 Blank.

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

- The programmer 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 RPG IV language resets the field or array element specified in factor 2 to blanks. Thus, if the programmer does not place a value in this field during the processing of the subroutine, the RPG IV 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, the programmer can specify the return point within the subroutine that is best suited for the exception/error that occurred. If the subroutine was called by the EXSR operation and factor 2 of the ENDSR operation is blank, 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/error occurs during the start or end of a program, control passes to the RPG IV 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/error occurs, an exception/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 will result in a program loop unless the programmer codes 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 on a halt indicator and issue the RETURN operation as follows:

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++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                                        H1
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
```

Figure 39. Setting a First-time Switch

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.

Program Exception/Errors

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

- 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.
- An indicator can be specified in positions 73 and 74 of the calculation specifications for some operation codes. This indicator is set on if an exception/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.

- 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.
- You can create a user-defined ILE exception handler that will take control when an exception occurs. For more information, see *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.
- A program exception/error subroutine can be specified. You enter *PSSR in factor 1 of a BEGSR operation to specify this subroutine. 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. You can also use the %STATUS built-in function, which returns the most recent value set for the program or file status.
- If the indicator, 'E' extender, monitor block, or program exception/error subroutine is not present, program exception/errors are handled by the RPG IV default error handler.

Program Status Data Structure

A program status data structure (PSDS) can be defined to make program exception/error information available to an RPG IV program. The PSDS must be defined in the main source section; therefore, there is only one PSDS per module.

A data structure is defined as a PSDS by an S in position 23 of the data structure statement. A PSDS contains predefined subfields that provide you with information about the program exception/error that occurred. The location of the subfields in the PSDS 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.

Information from the PSDS is also provided in a formatted dump. However, a formatted dump might not contain information for fields in the PSDS if the PSDS is not coded, or the length of the PSDS does not include those fields. For example, if the PSDS is only 275 bytes long, the time and date or program running will appear as *N/A*. in the dump, since this information starts at byte 276. For more information see "DUMP (Program Dump)" on page 669.

TIP

Call performance with LR on will be greatly improved by having no PSDS, or a PSDS no longer than 80 bytes, since some of the information to fill the PSDS after 80 bytes is costly to obtain.

Table 28 on page 98 provides the layout of the subfields of the data structure and the predefined From and To positions of its subfields that can be used to access information in this data structure.

Table 28. Contents of the Program Status Data Structure

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
1	10	Character	10	*PROC	If the module was compiled with CRTRPGMOD, this is the name of the module that was created; if the program was created using CRTBNDRPG, this is the name of the program that was created. For a cycle-main module, this is the name of the main procedure.
11	15	Zoned decimal	5,0	*STATUS	Status code. For a description of these codes, see "Program Status Codes" on page 101.
16	20	Zoned decimal	5,0		Previous status code.
21	28	Character	8		RPG IV source listing line number or statement number. The source listing line number is replaced by the source listing statement number if OPTION(*SRCSTMT) is specified instead of OPTION(*NOSRCSTMT). The full statement number is included when it applies to the root source member. If the statement number is greater than 6 digits (that is, it includes a source ID other than zero), the first 2 positions of the 8-byte feedback area will have a "+ " indicating that the rest of statement number is stored in positions 354-355.
29	36	Character	8	*ROUTINE	Name of the RPG IV routine in which the exception or error occurred. This subfield is updated at the beginning of an RPG IV 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 ***DETL** Detail lines ***GETIN** Get input record ***TOTC** Total calculations ***TOTL** Total lines ***DETC** Detail calculations ***OFL** Overflow lines ***TERM** Program ending ***ROUTINE** Name of program or procedure called (first 8 characters). **Note:** *ROUTINE is not valid unless you use the normal RPG IV cycle. Logic that takes the program out of the normal RPG IV cycle may cause *ROUTINE to reflect an incorrect value.

Table 28. Contents of the Program Status Data Structure (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
37	39	Zoned decimal	3,0	*PARMS	Number of parameters passed to this program from a calling program. The value is the same as that returned by %PARMS. If no information is available, -1 is returned.
40	42	Character	3		Exception type (CPF for an operating system exception or MCH for a machine exception).
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.
47	50	Character	4		Reserved
51	80	Character	30		Work area for messages. This area is only meant for internal use by the ILE RPG compiler. The organization of information will not always be consistent. It can be displayed by the user.
81	90	Character	10		Name of library in which the program is located.
91	170	Character	80		Retrieved exception data. CPF messages are placed in this subfield when location *STATUS contains 09999.
171	174	Character	4		Identification of the exception that caused RNX9001 exception to be signaled.
175	184	Character	10		Name of file on which the last file operation occurred (updated only when an error occurs). This information always contains the full file name.
185	190	Character	6		Unused.
191	198	Character	8		Date (*DATE format) the job entered the system. In the case of batch jobs submitted for overnight processing, those that run after midnight will carry the next day's date. This value is derived from the job date, with the year expanded to the full four years. 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). This file name will be truncated if a long file name is used. See positions 175-184 for long file name information.

Table 28. Contents of the Program Status Data Structure (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
209	243	Character	35		Status information on the last file used. This information includes the status code, the RPG IV opcode, the RPG IV routine name, the source listing line number or statement 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 The source listing line number is replaced by the source listing statement number if OPTION(*SRCSTMT) is specified instead of OPTION(*NOSRCSTMT). The full statement number is included when it applies to the root source member. If the statement number is greater than 6 digits (that is, it includes a source ID other than zero), the first 2 positions of the 8-byte feedback area will have a "+ " indicating that the rest of statement number is stored in positions 356-357.
244	253	Character	10		Job name.
254	263	Character	10		User name from the user profile.
264	269	Zoned decimal	6,0		Job number.
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	6,0		Time (in the format hhmmss) of the program running.
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.
314	323	Character	10		Source library name.
324	333	Character	10		Source file member name.
334	343	Character	10		Program containing procedure.

Table 28. Contents of the Program Status Data Structure (continued)

From (Pos. 26-32)	To (Pos. 33-39)	Format	Length	Keyword	Information
344	353	Character	10		Module containing procedure.
354	355	Binary	2		Source Id matching the statement number from positions 21-28.
356	357	Binary	2		Source Id matching the statement number from positions 228-235.
358	367	Character	10		Current user profile name.
368	371	Integer	10,0		External error code
372	379	Integer	20,0		XML elements set by operation
380	429	Character	50		Unused.

Program Status Codes

Any code placed in the subfield location *STATUS that is greater than 99 is considered to be an exception/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. Location *STATUS is updated when an exception/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

00001 Called program returned with the LR indicator on.

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 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; halt indicator (H1 through H9) not on

00211 Error calling program or procedure

00222 Pointer or parameter error

00231 Called program or procedure returned with halt indicator on

00232 Halt indicator on in this program

00233 Halt indicator on when RETURN operation run

00299 RPG IV formatted dump failed

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

00351 Error parsing XML document

00352 Invalid option for %XML

00353 XML document does not match RPG variable

00354 Error preparing for XML parsing

00401 Data area specified on IN/OUT not found

00402 *PDA not valid for non-prestart job

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

00425 Length requested for storage allocation is out of range

00426 Error encountered during storage management operation

00431 Data area previously locked by another program

00432 Data area locked by program in the same process

00450 Character field not entirely enclosed by shift-out and shift-in characters

00451 Conversion between two CCSIDs is not supported.

00501 Failure to retrieve sort sequence.

00502 Failure to convert 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)

00970 The level number of the compiler used to generate the program does not agree with the level number of the RPG IV run-time subroutines.

09998 Internal failure in ILE RPG compiler or in run-time subroutines

09999 Program exception in system routine.

PSDS Example

To specify a PSDS in your program, you 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                                             * Procedure 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 PGM_LIB               81     90                                     * Program library
D EXCP_DATA             91    170                                     * Exception data
D EXCP_ID              171    174                                     * Exception Id
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 JOB_NAME             244    253                                     * Job name
D USER                 254    263                                     * User name
D JOB_NUM              264    269S 0                                  * Job number
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 SRC_LIB              314    323                                     * Source file lib
D SRC_MBR              324    333                                     * Source file mbr
D PROC_PGM             334    343                                     * Pgm Proc is in
D PROC_MOD             344    353                                     * Mod Proc is in
```

Figure 40. 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/Error Subroutine

To identify the user-written RPG IV subroutine that is to receive control when a program exception/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 the 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 the operation code (that is, an array indexing error during a SCAN operation), control is transferred to this subroutine when a program exception/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 of the RPG IV 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 a discussion of the valid entries for factor 2, see "File Exception/Error Subroutine (INFSR)" on page 93.

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 RPG IV language resets the field, subfield, or array element specified in factor 2 to blanks. This allows you to specify the return point within the subroutine that is best suited for the exception/error that occurred. If factor 2 contains blanks at the end of the subroutine, the RPG IV 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.
- Because the program exception/error subroutine may receive control whenever a non-file exception/error occurs, an exception/error could occur while the subroutine is running. If an exception/error occurs while the subroutine is running, the subroutine is called again; this will result in a program loop unless the programmer codes the subroutine to avoid this problem.
- If you have used the OPTIMIZE(*FULL) option on either the CRTBNDRPG or the CRTRPGMOD command, you have to declare all fields that you refer to during exception handling with the NOOPT keyword in the definition specification for the field. This will ensure that when you run your program, the fields referred to during exception handling will have current values.
- 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.

Chapter 6. General File Considerations

This chapter contains a more detailed explanation of:
- Global and Local files
- File Parameters
- Variables Associated with Files
- Multi-file Processing
- Match fields
- Alternate collating sequence
- File translation.

Global and Local Files

In an RPG IV module, you can define global files which are available to every procedure in the module, or local files which are only available to one procedure. Global files are defined in the main source section, between the Control specifications and the Definition specifications. They can be primary, secondary, table, or full-procedural files. Local files are defined within subprocedures, between the Procedure specifications and the Definition specifications of the subprocedure. They can only be full-procedural files. Input and Output specifications can be defined to handle the field data for global files.

Input and Output specifications are not supported for subprocedures, so all input and output operations must be done using data structures for local files.

File Parameters

You can pass files as parameters using prototyped calls to RPG programs and procedures. You can define file parameters for prototypes and procedure interface definitions, using the LIKEFILE keyword. The called program or procedure can perform any operation that is valid on the original file that was used to define the file parameter.

Note: RPG file parameters are in a form that is not related to the forms used for file parameters in other languages such as C and C++. The file parameters used by RPG are not interchangeable with the file parameters used by other languages; you cannot pass a C file to an RPG procedure that is expecting an RPG file parameter, and you cannot pass an RPG file to a C program.

For an example of a program that passes a file parameter, see “Example of passing a file and passing a data structure with the associated variables.” on page 109

Variables Associated with Files

Using File specification keywords, you can associate several variables with a file. For example, the INFDS keyword associates a File Information Data Structure with the file; this data structure is updated by RPG during file operations with information about the current state of the file. The SFILE keyword defines a numeric variable that you set to the relative record number for a record that you are writing.

When a file is passed as a parameter, the file parameter in the called procedure continues to be associated with the same physical variables that it was associated with in the calling procedure. The called procedure has access to the associated variables of the file parameter, although this access is only available to the RPG compiler. This allows the RPG compiler to work with the associated variables when the called procedure performs operations on the file parameter. If a file operation to a file parameter requires the value of an associated variable, the current value of the associated variable will be used. If a file operation to a file parameter changes the contents of an associated variable, the associated variable will immediately be updated with the new value. Passing a file parameter does not give the called procedure direct access to the associated variables. The called procedure can only access the associated variables if they are global variables, or if they were passed as additional parameters to the procedure.

Tip: If you pass a file parameter to another procedure, and the procedure needs to be able to access the associated variables, define a data structure with a subfield for each associated variable, and pass that data structure as an additional parameter to the procedure. See Figure 41 on page 109. The following table lists the keywords that you can use to associate variables with a file.

Table 29. File specification keywords for associated variables

Keyword	Usage	Description
COMMIT	Input	The RPG programmer sets it to indicate whether the file is opened for commitment control.
DEVID	Input/Feedback	The RPG programmer sets it to direct file operations to a particular device. The RPG compiler sets it to indicate which device was used for the previous file operation.
EXTFILE	Input	The RPG programmer sets it to indicate the external file that is to be opened. The application developer sets it before the program is called to control whether a file is to be used. The RPG programmer sets it to indicate the external member that is to be opened.
EXTIND	Input	The RPG programmer sets some output-capable indicators for use by file operation. The system sets input-capable indicators during a operation.
EXTMBR	Input	The RPG compiler sets it to indicate the current state of a file.
INDDS	Input/Output	The RPG programmer sets some output-capable indicators for use by file operation. The system sets input-capable indicators during a operation
INFDS	Input	The RPG compiler sets it to indicate the current state of a file.
PRTCTL	Input/Feedback	The RPG programmer sets the space and skip fields to control the printer file.
RECNO	Input/Feedback	The RPG compiler sets it to indicate the current line of the printer file.
SAVEDS	Any	The RPG programmer sets it to indicate which relative record number is to be written to the subfile record.
SFILE	Input/Feedback	The RPG compiler sets it to indicate the relative record number that was retrieved by an input operation to the subfile record.
SLN	Input	The RPG programmer sets it to indicate the starting line for a display file record format.

Example of passing a file and passing a data structure with the associated variables.

The following example shows you how to define a data structure to hold the associated variables for a file, how to pass the file and the data structure as parameters to a procedure, and how to use the parameters within the procedure.

```
 * The /COPY file has template definitions for the File and Associated Variables

 /if defined(FILE_DEFINITIONS)
 // Template for the "INFILE" file type
Finfile_t  if   e            disk    template block(*yes)
F                                    extdesc('MYLIB/MYFILE')
 /eof
 /endif
 /if defined(DATA_DEFINITIONS)
 // Template for the associated variables for an INFILE file
D infileVars_t    ds                 qualified template
D    filename                  21a
D    mbrname                   10a
 // Prototype for a procedure to open an INFILE file
D open_infile     pr
D    theFile                         likefile(infile_t)
D    kwVars                          likeds(infileVars)
D                                    options(*nullind)
 /eof
 /endif
```

Figure 41. /COPY file INFILE_DEFS

```
P myproc          b
 // Copy in the template and prototype definitions
 /define FILE_DEFINITIONS
 /COPY INFILE_DEFS
 /undefine FILE_DEFINITIONS

 /define DATA_DEFINITIONS
 /COPY INFILE_DEFS
 /undefine DATA_DEFINITIONS
 // Define the file using LIKEFILE, to enable it to be passed as
 // a parameter to the "open_infile" procedure.

 // Define all the associated variables as subfields of a data
 // structure, so that all the associated variables can be
 // passed to the procedure as a single parameter
Ffile1                               likefile(infile_t)
F                                    extfile(file1Vars.filename)
F                                    extmbr(file1Vars.mbrname)
F                                    usropn

D file1Vars       ds                 likeds(infileVars_t)

  /free
        open_infile (file1 : file1Vars);

        . . .
```

Figure 42. The calling procedure that passes the file parameter

```
 // Copy in the template and prototype definitions
 /define FILE_DEFINITIONS
 /COPY INFILE_DEFS
 /undefine FILE_DEFINITIONS

 /define DATA_DEFINITIONS
 /COPY INFILE_DEFS
 /undefine DATA_DEFINITIONS
P open_infile     b
 // The open_infile procedure has two parameters
 //  - a file
 //  - a data structure containing all the associated variables for the file
D open_infile     pi
D    theFile                         likefile(infile_t)
D    kwVars                          likeds(infileVars)
  /free
         // The %OPEN(filename) built-in function reflects the
         // current state of the file
         if not %open(theFile);
            // The called procedure modifies the calling procedure's "file1Vars"
            // variables directly, through the passed parameter
            kwVars.extfile = 'LIB1/FILE1';
            kwVars.extmbr = 'MBR1';
            // The OPEN operation uses the file1Vars subfields in the
            // calling procedure to open the file, opening file LIB1/FILE1(MBR1)
            open theFile;
         endif;
         . . .
```

Figure 43. The called procedure that uses the file parameter

Primary/Secondary Multi-file Processing

In an RPG IV program, the processing of a primary input file and one or more secondary input files, with or without match fields, is termed multi-file processing. Selection of records from more than one file based on the contents of match fields is known as multi-file processing by matching records. Multi-file processing can be used with externally described or program described input files that are designated as primary/secondary files.

Multi-file Processing with No Match Fields

When no match fields are used in multi-file processing, records are selected from one file at a time. When the records from one file are all processed, the records from the next file are selected. The files are selected in this order:

1. Primary file, if specified
2. Secondary files in the order in which they are described on the file description specifications.

Multi-file Processing with Match Fields

When match fields are used in multi-file processing, the program selects the records for processing according to the contents of the match fields. At the beginning of the first cycle, the program reads one record from every primary/secondary input file and compares the match fields in the records. If the records are in ascending order, the program selects the record with the lowest match field. If the records are in descending order, the program selects the record with the highest match field.

When a record is selected from a file, the program reads the next record from that file. At the beginning of the next program cycle, the new record is compared with the other records in the read area that are waiting for selection, and one record is selected for processing.

Records without match fields can also be included in the files. Such records are selected for processing before records with match fields. If two or more of the records being compared have no match fields, selection of those records is determined by the priority of the files from which the records came. The priority of the files is:

1. Primary file, if specified
2. Secondary files in the order in which they are described on the file description specifications.

When the primary file record matches one or more of the secondary records, the MR (matching record) indicator is set on. The MR indicator is on for detail time processing of a matching record through the total time that follows the record. This indicator can be used to condition calculation or output operations for the record that is selected. When one of the matching records must be selected, the selection is determined by the priority of the files from which the records came.

Figure 9 on page 39 shows the logic flow of multi-file processing.

A program can be written where only one input file is defined with match fields and no other input files have match fields. The files without the match fields are then processed completely according to the previously mentioned priority of files. The file with the match fields is processed last, and sequence checking occurs for that file.

Assigning Match Field Values (M1-M9)

When assigning match field values (M1 through M9) to fields on the input specifications in positions 65 and 66, consider the following:

- Sequence checking is done for all record types with match field specifications. All match fields must be in the same order, either all ascending or all descending. The contents of the fields to which M1 through M9 are assigned are checked for correct sequence. An error in sequence causes the RPG IV exception/error handling routine to receive control. When the program continues processing, the next record from the same file is read.
- Not all files used in the program must have match fields. Not all record types within one file must have match fields either. However, at least one record type from two files must have match fields if files are ever to be matched.
- The same match field values must be specified for all record types that are used in matching. See Figure 44 on page 113.
- Date, time, and timestamp match fields with the same match field values (M1 through M9) must be the same type (for example, all date) but can be different formats.
- All character, graphic, or numeric match fields with the same match field values (M1 through M9) should be the same length and type. If the match field contains packed data, the zoned decimal length (two times packed length - 1) is used as the length of the match field. It is valid to match a packed field in one record against a zoned decimal field in another if the digit lengths are identical. The length must always be odd because the length of a packed field is always odd.

- Record positions of different match fields can overlap, but the total length of all fields must not exceed 256 characters.
- If more than one match field is specified for a record type, all the fields are combined and treated as one continuous field (see Figure 44 on page 113). The fields are combined according to descending sequence (M9 to M1) of matching field values.
- Match fields values cannot be repeated in a record.
- All match fields given the same matching field value (M1 through M9) are considered numeric if any one of the match fields is described as numeric.
- When numeric fields having decimal positions are matched, they are treated as if they had no decimal position. For instance 3.46 is considered equal to 346.
- Only the digit portions of numeric match fields are compared. Even though a field is negative, it is considered to be positive because the sign of the numeric field is ignored. Therefore, a -5 matches a +5.
- Date and time fields are converted to *ISO format for comparisons
- Graphic data is compared hexadecimally
- Whenever more than one matching field value is used, all match fields must match before the MR indicator is set on. For example, if match field values M1, M2, and M3 are specified, all three fields from a primary record must match all three match fields from a secondary record. A match on only the fields specified by M1 and M2 fields will not set the MR indicator on (see Figure 44 on page 113).
- UCS-2 fields cannot be used for matching fields.
- Matching fields cannot be used for lookahead fields, and arrays.
- Field names are ignored in matching record operations. Therefore, fields from different record types that are assigned the same match level can have the same name.
- If an alternate collating sequence or a file translation is defined for the program, character fields are matched according to the alternate sequence specified.
- Null-capable fields, character fields defined with ALTSEQ(*NONE), and binary, float, integer and unsigned fields (B, F, I, or U in position 36 of the input specifications) cannot be assigned a match field value.
- Match fields that have no field record relation indicator must be described before those that do. When the field record relation indicator is used with match fields, the field record relation indicator should be the same as a record identifying indicator for this file, and the match fields must be grouped according to the field record relation indicator.
- When any match value (M1 through M9) is specified for a field without a field record relation indicator, all match values used must be specified once without a field record relation indicator. If all match fields are not common to all records, a dummy match field should be used. Field record relation indicators are invalid for externally described files. (see Figure 45 on page 114).
- Match fields are independent of control level indicators (L1 through L9).
- If multi-file processing is specified and the LR indicator is set on, the program bypasses the multi-file processing routine.

Figure 44 on page 113 is an example of how match fields are specified.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
 * The files in this example are externally described (E in position
 * 22) and are to be processed by keys (K in position 34).
FMASTER    IP   E           K DISK
FWEEKLY    IS   E           K DISK
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IRcdname+++....Ri..........................................................
I.............Ext-field+.................Field+++++++++L1M1..PlMnZr....
 *                           MASTER FILE
IEMPMAS        01
I                                         EMPLNO         M1
I                                         DIVSON         M3
I                                         DEPT           M2
IDEPTMS        02
I                                         EMPLNO         M1
I                                         DEPT           M2
I                                         DIVSON         M3
 *                           WEEKLY FILE
IWEEKRC        03
I                                         EMPLNO         M1
I                                         DIVSON         M3
I                                         DEPT           M2
```

Figure 44. Match Fields in Which All Values Match

Three files are used in matching records. All the files have three match fields specified, and all use the same values (M1, M2, M3) to indicate which fields must match. The MR indicator is set on only if all three match fields in either of the files EMPMAS and DEPTMS are the same as all three fields from the WEEKRC file.

The three match fields in each file are combined and treated as one match field organized in the following descending sequence:

DIVSON M3

DEPT M2

EMPLNO M1

The order in which the match fields are specified in the input specifications does not affect the organization of the match fields.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IDISK      AB  01    1 C1
I         OR   02    1 C2
I         OR   03    1 C3
I                                 1   10 0EMPNO          M1
I                                11   15 0DUMMY          M2
I                                11   15 0DEPT           M202
I                                16   20 0DEPT           M203
```

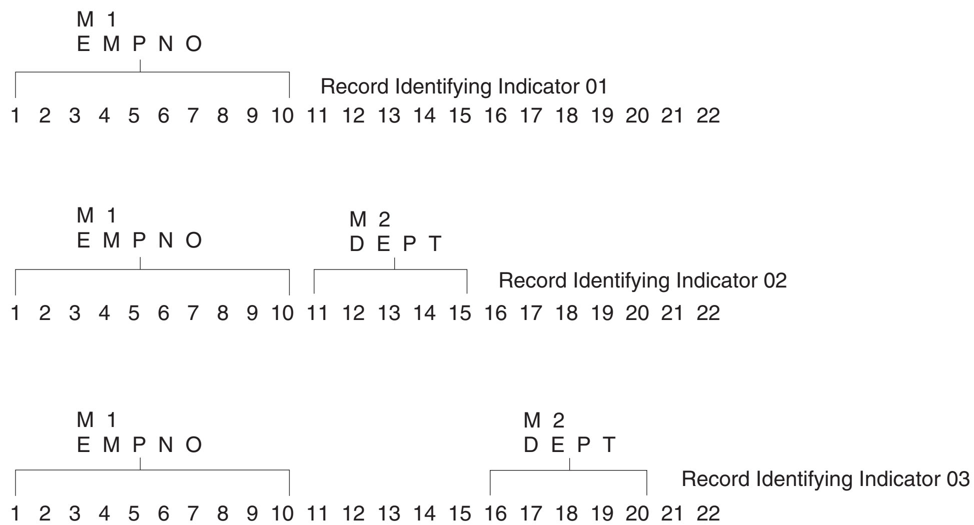


Figure 45. Match Fields with a Dummy M2 Field

Three different record types are found in the input file. All three contain a match field in positions 1 through 10. Two of them have a second match field. Because M1 is found on all record types, it can be specified without a field record relation entry in positions 67 and 68. If one match value (M1 through M9) is specified without field record relation entries, all match values must be specified once without field record relation entries. Because the value M1 is specified without field record relationship, an M2 value must also be specified once without field record relationship. The M2 field is not on all record types; therefore a dummy M2 field must be specified next. The dummy field can be given any unique name, but its specified length must be equal to the length of the true M2 field. The M2 field is then related to the record types on which it is found by field record relation entries.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
FPRIMARY   IPEA F   64        DISK
FFIRSTSEC  IS A F   64        DISK
FSECSEC    IS A F   64        DISK

*...1....+....2....+....3....+....4....+....5....+....6....+....7...
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IPRIMARY   AA  01    1 CP    2NC
I                                2    3  MATCH          M1
 *
I          BB  02    1 CP    2 C
I                                2    3  NOM
 *
IFIRSTSEC  AB  03    1 CS    2NC
I                                2    3  MATCH          M1
 *
I          BC  04    1 CS    2 C
I                                2    3  NOM
 *
ISECSEC    AC  05    1 CT    2NC
I                                2    3  MATCH          M1
 *
I          BD  06    1 CT    2 C
I                                2    3  NOM
```

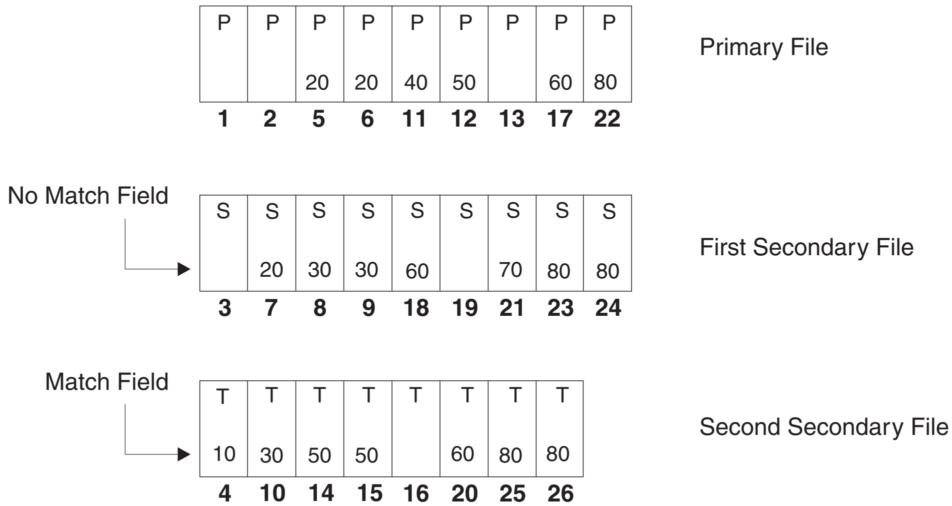
Figure 46. Match Field Specifications for Three Disk Files

Processing Matching Records

Matching records for two or more files are processed in the following manner:

- Whenever a record from the primary file matches a record from the secondary file, the primary file is processed first. Then the matching secondary file is processed. The record identifying indicator that identifies the record type just selected is on at the time the record is processed. This indicator is often used to control the type of processing that takes place.
- Whenever records from ascending files do not match, the record having the lowest match field content is processed first. Whenever records from descending files do not match, the record having the highest match field content is processed first.
- A record type that has no match field specification is processed immediately after the record it follows. The MR indicator is off. If this record type is first in the file, it is processed first even if it is not in the primary file.
- The matching of records makes it possible to enter data from primary records into their matching secondary records because the primary record is processed before the matching secondary record. However, the transfer of data from secondary records to matching primary records can be done only when look-ahead fields are specified.

Figure 47 on page 116 through Figure 48 on page 117 show how records from three files are selected for processing.



The records from the three disk files above are selected in the order indicated by the dark numbers.

Figure 47. Normal Record Selection from Three Disk Files

Table 30. Normal Record Selection from Three Disk Files

Cycle	File Processed	Indicators On	Reason for Setting Indicator
1	PRIMARY	02	No match field specified
2	PRIMARY	02	No match field specified
3	FIRSTSEC	04	No match field specified
4	SECSEC	05	Second secondary low; no primary match
5	PRIMARY	01, MR	Primary matches first secondary
6	PRIMARY	01, MR	Primary matches first secondary
7	FIRSTSEC	03, MR	First secondary matches primary
8	FIRSTSEC	03	First secondary low; no primary match
9	FIRSTSEC	03	First secondary low; no primary match
10	SECSEC	05	Second secondary low; no primary match
11	PRIMARY	01	Primary low; no secondary match
12	PRIMARY	01, MR	Primary matches second secondary
13	PRIMARY	02	No match field specified
14	SECSEC	05, MR	Second secondary matches primary
15	SECSEC	05, MR	Second secondary matches primary
16	SECSEC	06	No match field specified
17	PRIMARY	01, MR	Primary matches both secondary files
18	FIRSTSEC	03, MR	First secondary matches primary
19	FIRSTSEC	04	No match field specified
20	SECSEC	05, MR	Second secondary matches primary
21	FIRSTSEC	03	First secondary low; no primary match
22	PRIMARY	01, MR	Primary matches both secondary files
23	FIRSTSEC	03, MR	First secondary matches primary
24	FIRSTSEC	02, MR	First secondary matches primary

Table 30. Normal Record Selection from Three Disk Files (continued)

Cycle	File Processed	Indicators On	Reason for Setting Indicator
25	SECSEC	05, MR	Second secondary matches primary
26	SECSEC	05, MR	Second secondary matches primary

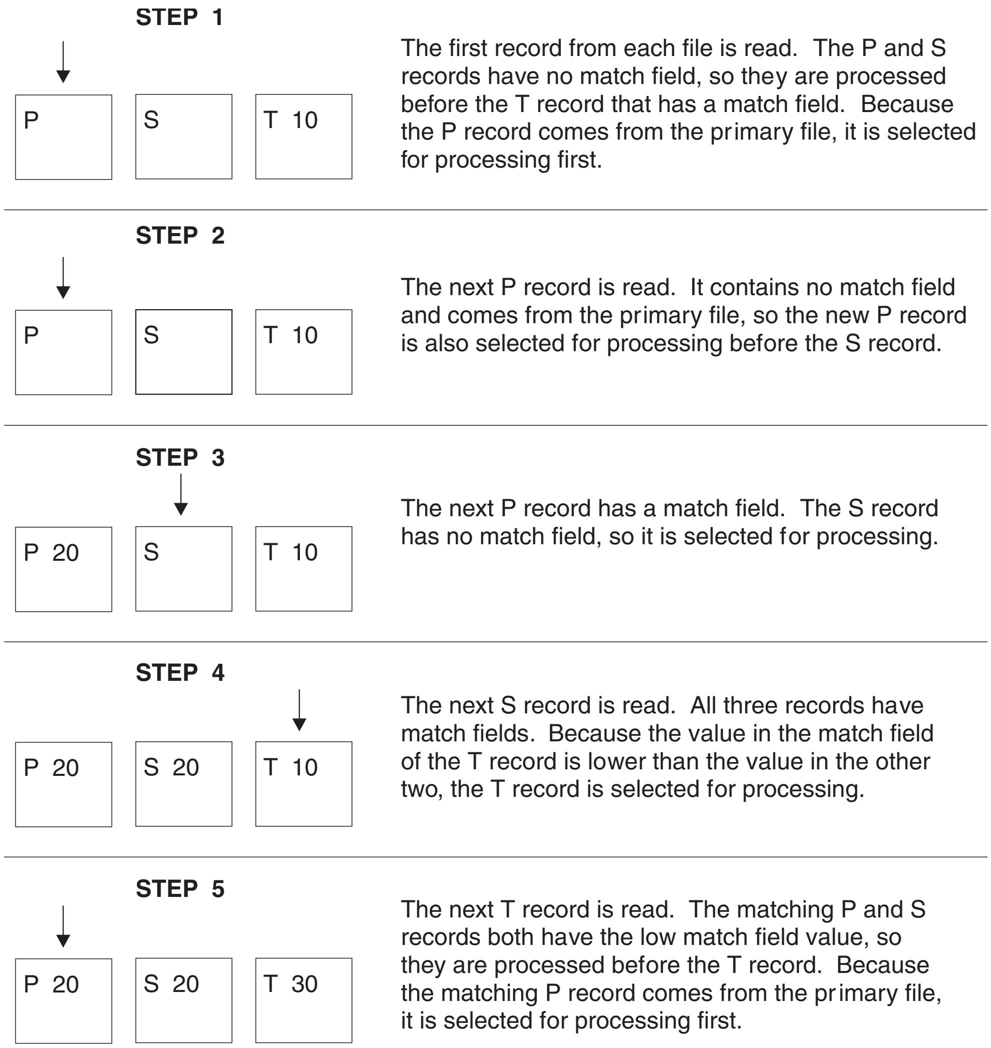


Figure 48. Normal Record Selection from Three Disk Files (Part 1 of 2)

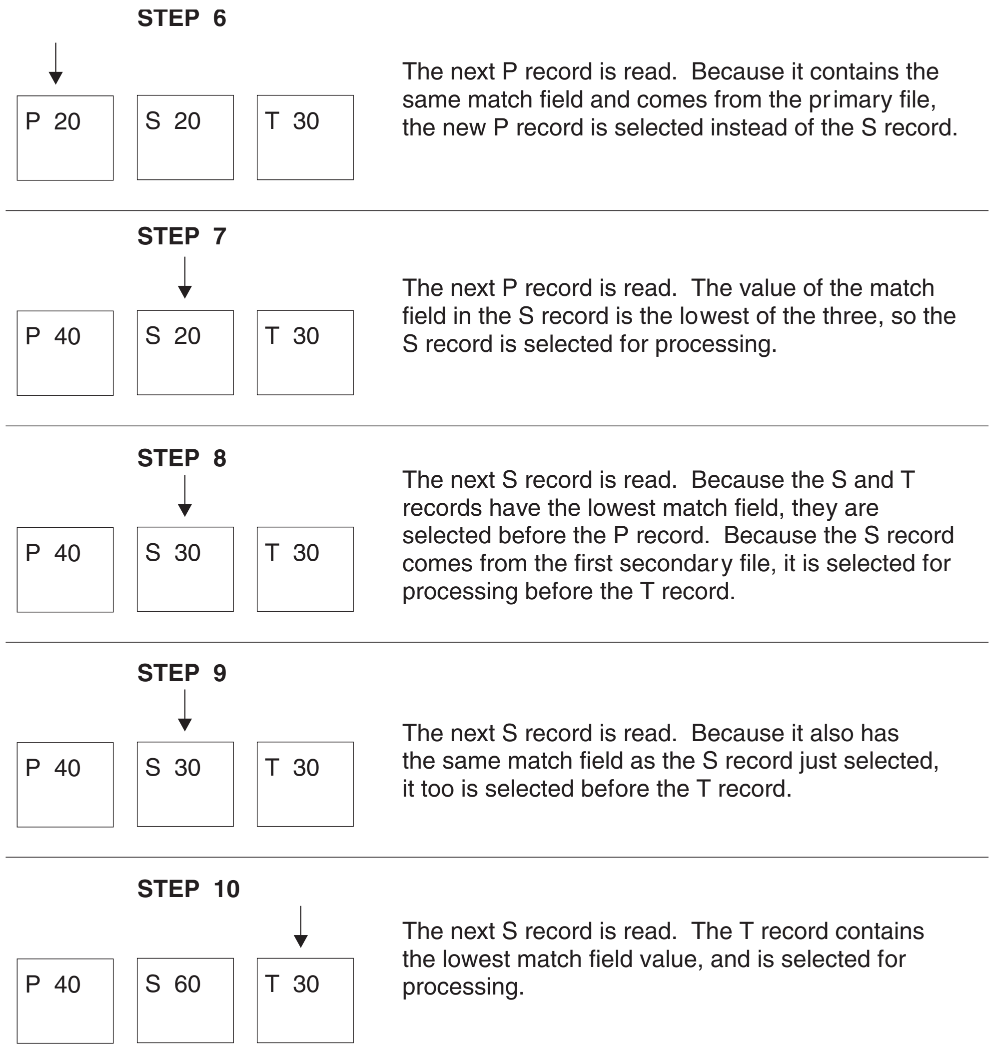


Figure 48. Normal Record Selection from Three Disk Files (Part 2 of 2)

File Translation

The file translation function translates any of the 8-bit codes used for characters into another 8-bit code. The use of file translation indicates one or both of the following:

- A character code used in the input data must be translated into the system code.
- The output data must be translated from the system code into a different code. The translation on input data occurs before any field selection has taken place. The translation on output data occurs after any editing taken place.

Remember the following when specifying file translation:

- File translation can be specified for data in array or table files (T in position 18 of the file description specifications).
- File translation can be used with data in combined, input, or update files that are translated at input and output time according to the file translation table provided. If file translation is used to translate data in an update file, each record must be written before the next record is read.

- For any I/O operation that specifies a search argument in factor 1 (such as CHAIN, READE, READPE, SETGT, or SETLL) for files accessed by keys, the search argument is translated before the file is accessed.
- If file translation is specified for both a record address file and the file being processed (if the file being processed is processed sequentially within limits), the records in the record address file are first translated according to the file translation specified for that file, and then the records in the file being processed are translated according to the file translation specified for that file.
- File translation applies only on a single byte basis.
- Every byte in the input and output record is translated.
- File translation is not supported for local files defined in subprocedures. #

Specifying File Translation

To specify file translation, use the FTRANS keyword on the control specification. The translations must be transcribed into the correct record format for entry into the system. These records, called the file translation table records, must precede any alternate collating sequence records, or arrays and tables loaded at compile time. They must be preceded by a record with **ƀ (ƀ = blank) in positions 1 through 3 or **FTRANS in positions 1 through 8. The remaining positions in this record can be used for comments.

Translating One File or All Files

File translation table records must be formatted as follows:

Record Position	Entry
1-8 (to translate all files)	Enter *FILESƀƀ (ƀ represents a blank) to indicate that all files are to be translated. Complete the file translation table record beginning with positions 11 and 12. If *FILESƀƀ is specified, no other file translation table can be specified in the program.
1-8 (to translate a specific file)	Enter the name of the file to be translated. Complete the file translation table record beginning with positions 11 and 12. The *FILESƀƀ entry is *not* made in positions 1 through 8 when a specific file is to be translated.
9-10	Blank
11-12	Enter the hexadecimal value of the character to be translated from on input or to be translated to on output.
13-14	Enter the hexadecimal equivalent of the internal character the RPG IV language works with. It will replace the character in positions 11 and 12 on input and be replaced by the character in positions 11 and 12 on output.
15-18 19-22 23-26 ... 77-80	All groups of four beginning with position 15 are used in the same manner as positions 11 through 14. In the first two positions of a group, enter the hexadecimal value of the character to be replaced. In the last two positions, enter the hexadecimal value of the character that replaces it.

The first blank entry ends the record. There can be one or more records per file translation table. When multiple records are required in order to define the table, the same file name must be entered on all records. A change in file name is used to separate multiple translation tables. An *FILES record causes all files, including tables and arrays specified by a T in position 18 of the file description specifications, to be translated by the same table.

```
     HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      * In this example all the files are translated
     H FTRANS
     FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++
     FFILE1     IP   F   10        DISK
     FFILE2     IS   F   10        DISK
     FFILE3     IS   F   10        DISK
     FFILE4     IS   F   10        DISK
**FTRANS
*FILES    81C182C283C384C4
```

```
     HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      * In this example different translate tables are used and
      * FILE3 is not translated.
     H FTRANS
     FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++
     FFILE1     IP   F   10        DISK
     FFILE2     IS   F   10        DISK
     FFILE3     IS   F   10        DISK
     FFILE4     IS   F   10        DISK
**FTRANS
FILE1     8182
FILE2     C1C2
FILE4     81C182C283C384C4
```

Translating More Than One File

If the same file translation table is needed for more than one file but not for all files, two types of records must be specified. The first record type specifies the file using the tables, and the second record type specifies the table. More than one record for each of these record types can be specified. A change in file names is used to separate multiple translation tables.

Specifying the Files

File translation table records must be formatted as follows:

Record Position	Entry
1-7	*EQUATE
8-10	Leave these positions blank.
11-80	Enter the name(s) of file(s) to be translated. If more than one file is to be translated, the file names must be separated by commas.

Additional file names are associated with the table until a file name not followed by a comma is encountered. A file name cannot be split between two records; a comma following a file name must be on the same record as the file name. You can create only one file translation table by using *EQUATE.

Specifying the Table

File translation table records must be formatted as follows:

Record Position	Entry
1-7	*EQUATE

Record Position	Entry
8-10	Leave these positions blank.
11-12	Enter the hexadecimal value of the character to be translated from on input or to be translated to on output.
13-14	Enter the hexadecimal equivalent of the internal character the RPG IV language works with. It will replace the character in positions 11 and 12 on input and be replaced by the character in positions 11 and 12 on output.
15-18 19-22 23-26 ... 77-80	All groups of four beginning with position 15 are used the same way as positions 11 through 14. In the first two positions of a group, enter the hexadecimal value of the character to be replaced. In the last two positions, enter the hexadecimal value of the character that replaces it.

The first blank record position ends the record. If the number of entries exceeds 80 positions, duplicate positions 1 through 10 on the next record and continue as before with the translation pairs in positions 11 through 80. All table records for one file must be kept together.

The records that describe the file translation tables must be preceded by a record with **ƀ (ƀ = blank) in positions 1 through 3 or with **FTRANS. The remaining positions in this record can be used for comments.

```
     HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      * In this example several files are translated with the
      * same translation table.  FILE2 is not translated.
     H FTRANS
     FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++
     FFILE1     IP   F   10        DISK
     FFILE2     IS   F   10        DISK
     FFILE3     IS   F   10        DISK
     FFILE4     IS   F   10        DISK
**FTRANS
*EQUATE   FILE1,FILE3,FILE4
*EQUATE   81C182C283C384C485C586C687C788C889C98ACA8BCB8CCC8DCD8ECE8F
*EQUATE   91D192D2
```

Part 2. Definitions

This section provides information on the different types of definitions that can be coded in your source. It describes:

- How to define
  - Standalone fields, arrays, and tables
  - Named constants
  - Data structures and their subfields
  - Prototypes
  - Prototyped parameters
  - Procedure interface
- Scope and storage of definitions as well as how to define each definition type.
- Data types and Data formats
- Editing numeric fields

For information on how to define files, see Chapter 13, "File Description Specifications," on page 279 and also the chapter on defining files in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Chapter 7. Defining Data and Prototypes

ILE RPG allows you to define the following items:

- Data items such as data structures, data-structure subfields, standalone fields, and named constants. Arrays and tables can be defined as either a data-structure subfield or a standalone field.
- Prototypes, procedure interfaces, and prototyped parameters

This chapter presents information on the following topics:

- General considerations, including definition types, scope, and storage
- Standalone fields
- Constants
- Data Structures
- Prototypes, parameters, and procedure interfaces

General Considerations

You define items by using definition specifications. Definitions can appear in two places within a module or program: within the cycle-main source section and within a subprocedure. (The **main source section** consists of the first set of H, F, D, I, C, and O specifications in a module; it corresponds to the specifications found in a standalone program or a cycle-main procedure.) 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.

```
 *-----------------------------------------------------------------*
 *  Global Definitions
 *-----------------------------------------------------------------*
D String          S              6A   INZ('ABCDEF')
D Spcptr          S               *
D SpcSiz          C                   8
D DS1             DS                  OCCURS(3)
D  Fld1                          5A   INZ('ABCDE')
D  Fld1a                         1A   DIM(5) OVERLAY(Fld1)
D  Fld2                          5B 2 INZ(123.45)
D Switch          PR
D   Parm                         1A
...
 *-----------------------------------------------------------------*
 *  Local Definitions
 *-----------------------------------------------------------------*
P Switch          B
D Switch          PI
D   Parm                         1A
 * Define a local variable.
D Local           S              5A      INZ('aaaaa')
...
P                 E
```

Figure 49. Sample Definitions

Scope of Definitions

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, as shown in Figure 50.

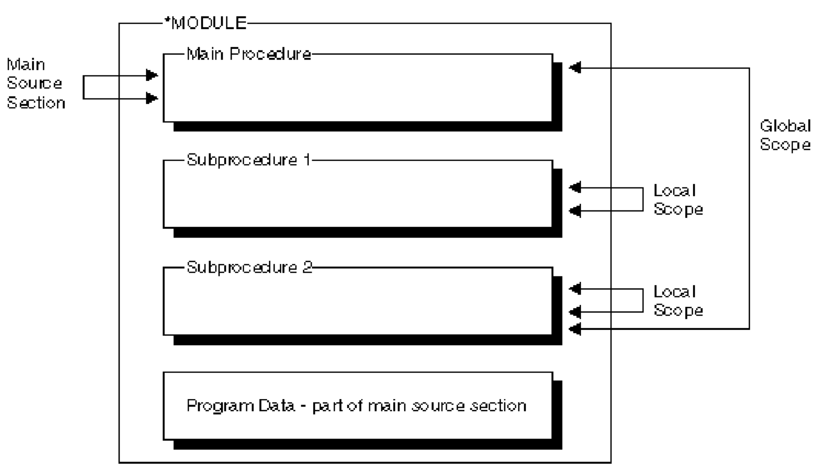


Figure 50. Scope of Definitions

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 cycle-main procedure and any subprocedures within the module. They can also be exported.

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 are defined in the cycle-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.

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 24.

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.

In a cycle module, static storage for global definitions is subject to the RPG cycle, and so the value changes on the next call to the cycle-main procedure if LR was on at the end of the last call. However, local static variables will not get reinitialized because of LR in the cycle-main procedure.

> **TIP**
>
> 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.

Standalone Fields

Standalone fields allow you to define individual work fields. A standalone field has the following characteristics:

- It has a specifiable internal data type
- It may be defined as an array, table, or field
- It is defined in terms of data length, not in terms of absolute byte positions.

For more information on standalone fields, see:
- Chapter 8, "Using Arrays and Tables," on page 159
- Chapter 9, "Data Types and Data Formats," on page 179
- "Definition-Specification Keywords" on page 321

Variable Initialization

You can initialize data with the "INZ{(initial value)}" on page 338 keyword on the definition specification. Specify an initial value as a parameter on the INZ keyword, or specify the keyword without a parameter and use the default initial values. If the initialization is too complicated to express using the INZ keyword, you can further initialize data in the initialization subroutine.

Default initial values for the various data types are described in Chapter 9, "Data Types and Data Formats," on page 179. See Chapter 8, "Using Arrays and Tables," on page 159 for information on initializing arrays.

To reinitialize data while the program is running, use the CLEAR and RESET operations.

The CLEAR operation code sets a record format or variable (field, subfield, indicator, data structure, array, or table) to its default value. All fields in a record format, data structure, or array are cleared in the order in which they are declared.

The RESET operation code restores a variable to its reset value. The reset value for a global variable is the value it had at the end of the initialization step in the RPG IV cycle, after the initialization subroutine has been invoked.

You can use the initialization subroutine to assign initial values to a global variable and then later use RESET to set the variable back to this value. This applies only to the initialization subroutine when it is run automatically as a part of the initialization step.

For local variables the reset value is the value of the variable when the subprocedure was first called, but before the calculations begin.

Constants

Literals and named constants are types of constants. They 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
- As the format name in a WORKSTN output specification
- 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 RPG IV data types.

Character Literals

The following are the rules for specifying a character literal:

- Any combination of characters can be used in a character literal. This includes DBCS characters. DBCS characters must be enclosed by shift-out and shift-in characters and must be an even number of bytes. Embedded blanks are valid.
- A character literal with no characters between the apostrophes is allowed. See Figure 52 on page 133 for examples.
- 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 are the rules for specifying a hexadecimal literal:

- Hexadecimal literals take the form:

  ```
  X'x1x2...xn'
  ```

  where `X'x1x2...xn'` can only 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.
- Except when used in the bit operations BITON, BITOFF, and TESTB, a hexadecimal literal has the same meaning as the corresponding character literal. 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'C17DC2'` not `X'C17D7DC2'`.
- 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 are the rules for 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 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.

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'xx-xx-xx' where:
- D indicates that the literal is of type date
- xx-xx-xx is a valid date in the format specified on the control specification (separator included)
- xx-xx-xx is enclosed by apostrophes

Time Literals

Time literals take the form T'xx:xx:xx' where:
- T indicates that the literal is of type time
- xx:xx:xx is a valid time in the format specified on the control specification (separator included)

- xx:xx:xx 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
- Microseconds are optional and if not specified will default to zeros

Graphic Literals

Graphic literals take the form G'oK1K2i' where:
- G indicates that the literal is of type graphic
- o is a shift-out character
- K1K2 is an even number of bytes (possibly zero) and does not contain a shift-out or shift-in character
- i is a shift-in character
- oK1K2i 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.

Example of Defining Literals

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
H DATFMT(*ISO)
 * Examples of literals used to initialize fields
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.....................................Keywords+++++++++++++++++++++++++++++
D DateField       S               D   INZ(D'1988-09-03')
D NumField        S              5P 1 INZ(5.2)
D CharField       S             10A   INZ('abcdefghij')
D UCS2Field       S              2C   INZ(U'00610062')
 * Even though the date field is defined with a 2-digit year, the
 * initialization value must be defined with a 4-digit year, since
 * all literals must be specified in date format specified
 * on the control specification.
D YmdDate         S               D   INZ(D'2001-01-13')
D                                     DATFMT(*YMD)
 * Examples of literals used to define named constants
D DateConst       C                   CONST(D'1988-09-03')
D NumConst        C                   CONST(5.2)
D CharConst       C                   CONST('abcdefghij')

 * Note that the CONST keyword is not required.
D Upper           C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

 * Note that the literal may be continued on the next line
D Lower           C                   'abcdefghijklmn-
D                                     opqrstuvwxyz'

 * Examples of literals used in operations
C                   EVAL      CharField = 'abc'
C                   IF        NumField > 12
C                   EVAL      DateField = D'1995-12-25'
C                   ENDIF
```

Figure 51. Defining named constants

Example of Using Literals with Zero Length

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.......................................Keywords+++++++++++++++++++++++++++++
 * The following two definitions are equivalent:
D varfld1         S              5    INZ VARYING
D varfld2         S              5    INZ('') VARYING
 * Various fields used by the examples below:
D blanks          S             10    INZ
D vblanks         S             10    INZ('   ') VARYING
D fixfld1         S              5    INZ('abcde')
 * VGRAPHIC and VUCS2 are initialized with zero-length literals.
D vgraphic        S             10G   INZ(G'oi') VARYING
D vucs2           S             10C   INZ(U'') VARYING
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq++++
 * The following statements do the same thing:
C                   eval      varfld1 = ''
C                   clear                   varfld1
 * Moving '' to a variable-length field using MOVE(P) or MOVEL(P)
 * sets the field to blanks up to the fields current length.
C                   move(p)   ''            varfld1
C                   movel(p)  ''            varfld1

 * Moving '' to a fixed-length field has no effect in the following
 * examples:  (The rightmost or leftmost 0 characters are changed.)
C                   move      ''            fixfld1
C                   movel     ''            fixfld1

 * The following comparisons demonstrate how the shorter operand
 * is padded with blanks:
C                   eval      *in01 = (blanks = '')
 * *in01 is '1'

C                   eval      *in02 = (vblanks = '')
 * *in02 is '1'

C                   eval      *in03 = (varfld2 = blanks)
 * *in03 is '1'

C                   eval      *in04 = (varfld2 = vblanks)
 * *in04 is '1'

C                   eval      *in05 = (%len(vgraphic)=0)
 * *in05 is '1'

 C                  eval      *in06 = (%len(vucs2)=0)
 * *in06 is '1'
```

Figure 52. Character, Graphic, and UCS-2 Literals with Zero Length

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. Numeric named constants have no predefined precision. Their actual precision is defined by the context that is specified.

See Figure 51 on page 132 for examples of defining named constants. The value of the named constant is specified in the keyword section of the definition specification. The presence of the keyword CONST is optional, however. For example, to assign a value of 'ab' to a constant, you could specify either CONST('ab') or 'ab' in the keyword section.

Figurative Constants

The figurative constants *BLANK/*BLANKS, *ZERO/*ZEROS, *HIVAL, *LOVAL, *NULL, *ALL'x..', *ALLG'oK1K2i', *ALLU'XxxxYyyy', *ALLX'x1..', and *ON/*OFF 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. (For exceptions, see the following section, "Rules for Figurative Constants" on page 135.)

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. The value for character is ' ' (blank) or X'40', for graphic is X'4040', and for UCS-2 is X'0020'.

***ZERO/*ZEROS**
: **Character/numeric fields**: All zeros. The value is '0' or X'F0'. **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**: The maximum value allowed for the corresponding field (with a positive sign if applicable). **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 Type" on page 206, "Time Data Type" on page 208 and "Timestamp Data Type" on page 210 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**: The minimum value allowed (with a negative sign if applicable). **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 Type" on page 206, "Time Data Type" on page 208 and "Timestamp Data Type" on page 210 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.

  Note: For numeric integer or unsigned fields, the value is never greater than the maximum value allowed for the corresponding field. For example, *ALL'95' represents the value 9595 if the corresponding field is a 5-digit integer field, since 95959 is greater than the maximum value allowed for a 5-digit signed integer.

***ALLG'oK1K2i'**
: **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 is all ones ('1' or X'F1'). *OFF is all zeros ('0' or X'F0'). Both are only valid for character fields.

Rules for Figurative Constants

Remember the following rules when using figurative constants:

- MOVE and MOVEL operations allow you to move 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, then 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. If not, a decimal data error will occur.
- 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 will contain '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...'.

Note that the results of MOVEA are different from those of the MOVE example above.

- 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'oK1K2i', 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.

Data Structures

The ILE RPG compiler allows you to define an area in storage and the layout of the fields, called subfields, within the area. This area in storage is called a **data structure**. You define a data structure by specifying DS in positions 24 through 25 on a definition specification.

You can use a data structure to:
- Define the same internal area multiple times using different data formats
- 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.
- Operate on all the subfields as a group using the name of the data structure.
- Operate on an individual subfield using its name.

In addition, there are four special data structures, each with a specific purpose:
- A data area data structure (identified by a U in position 23 of the definition specification)
- A file information data structure (identified by the keyword INFDS on a file description specification)
- A program-status data structure (identified by an S in position 23 of the definition specification)
- An indicator data structure (identified by the keyword INDDS on a file description specification).

Data structures can be either program-described or externally described, except for indicator data structures, which are program-described only. 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.

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. At compile time, the ILE RPG compiler uses the external name to locate and extract the external description of the data structure subfields. You specify the name of the external file either in positions 7 through 21, or as a parameter for the keyword EXTNAME .

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.

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 most free-form calculation specifications, or in any Extended-Factor-2 entry. This includes operation 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 363 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. Expressions are allowed as Factor 1 and Factor 2 operands 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.

A **"Keyed Array Data Structure"** is an array data structure with one subfield identified as the search or sort key. The array data structure is indexed by (*) and followed by the specification of the key subfield. For example, consider array data structure FAMILIES with one scalar subfield NAME, and another array subfield CHILDREN. To use the FAMILIES data structure as an array data structure keyed by NAME, specify FAMILIES(*).NAME. To use the first CHILDREN element as the key, specify FAMILIES(*).CHILDREN(1).

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 * except in the context of a keyed array data structure. Within a fully qualified name expression, an array index may only be omitted (or * specified) for the right-most name.
5. An array data structure can be sorted using the “SORTA (Sort an Array)” on page 815 operation code. The array is sorted using one of the subfields as a key.
6. An array data structure can be searched using the %LOOKUP built-in function. The array is searched using one of the subfields as a key.
7. Here are some examples of statements using keyed array data structure expressions that are not valid. Assume that TEAMS is an array data structure with scalar subfield MANAGER and data structure subfield EMPS.
   a. These statements are not valid because TEAMS is an array data structure. A non-array key subfield must be specified.
      ```
      SORTA TEAMS;
      SORTA TEAMS(*);
      ```
   b. These statements are not valid because TEAMS(1).EMPS is an array data structure. A non-array key subfield must be specified.
      ```
      SORTA TEAMS(1).EMPS;
      SORTA TEAMS(1).EMPS(*);
      ```
   c. This statement is not valid because TEAMS(*).EMPS(*) specifies two different arrays to be sorted. Only one (*) may be specified.
      ```
      SORTA TEAMS(*).EMPS(*).NAME;
      ```
   d. These statements are not valid because all arrays in the qualified name must be indexed. Both the TEAMS and the EMPS subfields must be indexed; one must be indexed with (*).
      ```
      SORTA TEAMS(*).EMPS.NAME;
      SORTA TEAMS.EMPS(*).NAME;
      ```
   e. This statement is not valid because at least one array must be indexed by (*). TEAMS(1).EMPS(1).NAME is a scalar value.
      ```
      SORTA TEAMS(1).EMPS(1).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 131 on page 341 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. See Figure 57 on page 146 for an example using the OVERLAY keyword.

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 142.

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 run-time 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 53 shows a sequence of bytes and identifies the different boundaries used for alignment.

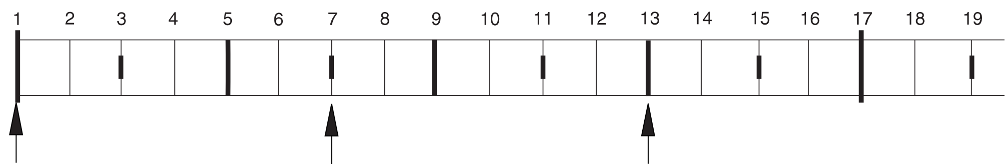


Figure 53. Boundaries for Data Alignment

Note the following about the above 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
- Indicator data structures.

Note that the above data structures cannot be defined in subprocedures.

Data Area Data Structure

A data area data structure, identified by a U in position 23 of the definition specification, indicates to the compiler that it should read in and lock the data area of the same name at program initialization and should write out and unlock the same data area at the end of the program. Locking does not apply to the local data area (see “Local Data Area (*LDA)”). Data area data structures, as in all 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 within the ILE RPG program by using the *DTAARA DEFINE operation code or the DTAARA keyword.

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 data structure name in the result field of the *DTAARA DEFINE operation or with the DTAARA keyword.

A data area data structure cannot be specified in the result field of a PARM operation in the *ENTRY PLIST.

Local Data Area (*LDA): If you specify blanks for the data area data structure (positions 7 through 21 of the definition specification that contains a U in position 23), the compiler uses the local data area. To provide a name for the local data area, use the *DTAARA DEFINE operation, with *LDA in factor 2 and the name in the result field or DTAARA(*LDA) on the definition specification.

File Information Data Structure

You can specify a file information data structure (defined by the keyword INFDS on a file description specifications) for each file in the program. This provides you with status information on the file exception/error that occurred. A file information data structure can be used for only one file. A file information data structure contains predefined subfields that provide information on the file exception/error that occurred. For a discussion of file information data structures and their subfields, see “File Information Data Structure” on page 79.

Program-Status Data Structure

A program-status data structure, identified by an S in position 23 of the definition specification, provides program exception/error information to the program. For a discussion of program-status data structures and their predefined subfields, see “Program Status Data Structure” on page 97.

Indicator Data Structure

An indicator data structure is identified by the keyword INDDS on the file description specifications. It is used to store conditioning and response indicators passed to and from data management for a file. By default, the indicator data structure is initialized to all zeros ('0's).

The rules for defining the data structure are:

- It must not be externally described.
- It can only have indicator or fixed-length character subfields.
- It can be defined as a multiple occurrence data structure.
- %SIZE for the data structure will return 99. For a multiple occurrence data structure, %SIZE(ds:*ALL) will return a multiple of 99. If a length is specified, it must be 99.
- Subfields may contain arrays of indicators as long as the total length does not exceed 99.

Data Structure Examples

The following examples show various uses for data structures and how to define them.

Example	Description
Figure 54	Using a data structure to subdivide a field
Figure 55 on page 144	Using a data structure to group fields
Figure 56 on page 145	Using keywords QUALIFIED, LIKEDS, and DIM with data structures, and how to code fully-qualified subfields
Figure 57 on page 146	Data structure with absolute and length notation
Figure 58 on page 146	Rename and initialize an externally described data structure
Figure 59 on page 147	Using PREFIX to rename all fields in an external data structure
Figure 60 on page 147	Defining a multiple occurrence data structure
Figure 61 on page 148	Aligning data structure subfields
Figure 62 on page 149	Defining a *LDA data area data structure
Figure 63 on page 150	Using data area data structures (1)
Figure 64 on page 150	Using data area data structures (2)
Figure 65 on page 151	Using an indicator data structure
Figure 66 on page 152	Using a multiple-occurrence indicator data structure

```
*.. 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++SqNORiPos1+NCCPos2+NCCPos3+NCC.................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr......
 *
 *  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 54. 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++SqNORiPos1+NCCPos2+NCCPos3+NCC..................................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr......
 *
 *  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
CL0N01Factor1+++++++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 55. 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 56. 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 57. 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 the DIM keyword.
 *  The ITMARR subfield is defined in the external description as a
 *  100 byte character field.  This divides the 100 byte character
 *  field into 10 array elements, each 10 bytes long.
 *  Using the DIM keyword on an externally described numeric subfield
 *  should be done with caution, because it will divide the field into
 *  array elements (similar to the way it does when absolute notation
 *  is used for program described subfields).
 *
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 58. 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 59. 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 'Max_Occur' is used to define the number of
 *  occurrences.
 *
 *  Absolute notation (using begin/end positions)
 *
D Max_Occur       C                   CONST(20)
D
DDataStruct       DS                  OCCURS (Max_Occur)
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 60. 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 61. Aligning Data Structure Subfields

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.....................................Keywords+++++++++++++++++++++++++++++
 *
 *  Define a data area data structure based on the *LDA.
 *
 *  Example 1:
 *  A data area data structure with no name is based on the *LDA.
 *  In this case, the DTAARA keyword does not have to be used.
 *
D                UDS
D  SUBFLD                 1    600A
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *  Example 2:
 *  This data structure is explicitly based on the *LDA using
 *  the DTAARA keyword.  Since it is not a data area data
 *  structure, it must be handled using IN and OUT operations.
 *
D LDA_DS          DS                  DTAARA(*LDA)
D  SUBFLD                 1    600A
...
C                   IN        LDA_DS
C                   OUT       LDA_DS
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *  Example 3:
 *  This data structure is explicitly based on the *LDA using
 *  the DTAARA keyword.  Since it is a data area data
 *  structure, it is read in during initialization and written
 *  out during termination.  It can also be handled using IN
 *  and OUT operations, since the DTAARA keyword was used.
 *
D LDA_DS         UDS                  DTAARA(*LDA)
D  SUBFLD                 1    600A
...
C                   IN        LDA_DS
C                   OUT       LDA_DS
```

*Figure 62. Defining a *LDA data area data structure*

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DFTNAME(Program1)
H
 *
FFilename++IPEASF.....L.....A.Device+.Keywords+++++++++++++++++++++++++++
FSALESDTA  IF   E             DISK
 *
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D...................................Keywords+++++++++++++++++++++++++++++
 *
 *  This program uses a data area data structure to accumulate
 *  a series of totals.  The data area subfields are then added
 *  to fields from the file SALESDTA.
D  Totals        UDS
D    Tot_amount                 8 2
D    Tot_gross                 10 2
D    Tot_net                   10 2
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++++++++++++++++++++++++++++++++
 *
C                   :
C                   EVAL      Tot_amount = Tot_amount + amount
C                   EVAL      Tot_gross  = Tot_gross  + gross
C                   EVAL      Tot_net    = Tot_net    + net
```

Figure 63. Using data area data structures (program 1)

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DFTNAME(Program2)
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D...................................Keywords+++++++++++++++++++++++++++++
 *
 *  This program processes the totals accumulated in Program1.
 *  Program2 then uses the total in the subfields to do calculations.
 *
D  Totals        UDS
D    Tot_amount                 8 2
D    Tot_gross                 10 2
D    Tot_net                   10 2
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C                   :
C                   EVAL      *IN91 =  (Amount2 <> Tot_amount)
C                   EVAL      *IN92 =  (Gross2 <> Tot_gross)
C                   EVAL      *IN93 =  (Net2   <> Tot_net)
C                   :
```

Figure 64. Using data area data structures (program 2)

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
FFilename++IPEASFRLen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 * Indicator data structure "DispInds" is associated to file "Disp".
FDisp      CF   E             WORKSTN INDDS (DispInds)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.....................................Keywords+++++++++++++++++++++++++++++
 *
 * This is the indicator data structure:
 *
D DispInds        DS
 * Conditioning indicators for format "Query"
D    ShowName             21    21N
 * Response indicators for format "Query"
D    Exit                  3     3N
D    Return               12    12N
D    BlankNum             31    31N
 * Conditioning indicators for format "DispSflCtl"
D    SFLDSPCTL            41    41N
D    SFLDSP               42    42N
D    SFLEND               43    43N
D    SFLCLR               44    44N
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * Set indicators to display the subfile:
C                   EVAL      SFLDSP = *ON
C                   EVAL      SFLEND = *OFF
C                   EVAL      SFLCLR = *OFF
C                   EXFMT     DispSFLCTL
 *
 * Using indicator variables, we can write more readable programs:
C                   EXFMT     Query
C                   IF        Exit or Return
C                   RETURN
C                   ENDIF
```

Figure 65. Using an indicator data structure

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
FFilename++IPEASFRLen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
 * Indicator data structure "ErrorInds" is associated to file "Disp".
FDisp      CF   E             WORKSTN INDDS (ERRORINDS)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.....................................Keywords+++++++++++++++++++++++++++++
D @NameOk         C                   0
D @NameNotFound   C                   1
D @NameNotValid   C                   2
D @NumErrors      C                   2
 *
 * Indicator data structure for ERRMSG:
 *
D ERRORINDS        DS                  OCCURS(@NumErrors)
 * Indicators for ERRMSG:
D   NotFound               1     1N
D   NotValid               2     2N
 *
 * Indicators for QUERY:
D    Exit                  3     3N
D    Refresh               5     5N
D    Return               12    12N
 *
 * Prototype for GetName procedure (code not shown)
D GetName         PR            10I 0
D    Name                       50A    CONST
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C                   DOU       Exit or Return
C                   EXFMT     QUERY
 * Check the response indicators
C                   SELECT
C                   WHEN      Exit or Return
C                   RETURN
C                   WHEN      Refresh
C                   RESET            QUERY
C                   ITER
C                   ENDSL
 *
 * Check the name
C                   EVAL      RC = GetName(Name)
 *
 * If it is not valid, display an error message
C                   IF        RC <> @NameOk
C     RC            OCCURS    ErrorInds
C                   EXFMT     ERRMSG
C                   ENDIF
C                   ENDDO
 ...
C     *INZSR        BEGSR
 *
 * Initialize the occurrences of the ErrorInds data structure
C     @NameNotFound OCCUR     ErrorInds
C                   EVAL      NotFound = '1'
C     @NameNotValid OCCUR     ErrorInds
C                   EVAL      NotValid = '1'
C                   ENDSR
```

Figure 66. Using a multiple-occurrence indicator data structure

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 these concepts:
- "Prototypes"
- "Prototyped Parameters" on page 155
- "Procedure Interface" on page 157.

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)
- The number and nature of the parameters
- Which parameters must be passed, and which are optionally passed
- Whether operational descriptors should be passed
- The data type of the return value, if any (for a procedure)

A prototype may be explicitly or implicitly defined. If the procedure is called from a different RPG module, the prototype must be explicitly specified in both the calling module and the module that defines the procedure. If the procedure is only called within the same module, the prototype may be explicitly defined, or it may be omitted. If the prototype is omitted, the compiler will implicitly define it from the procedure interface.

For modules that call a procedure that is defined in a different module, 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)
The call will be an external program call that uses the external name specified by the keyword.

OPDESC
Operational descriptors are to be passed with the parameters that are described in the prototype.

RTNPARM
The return value is to be handled as a parameter. This may improve performance when calling the procedure, especially for large return values.

- 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.

  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.

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

  VARYING{(2|4)}
  A character, graphic, or UCS-2 return value has a variable-length format.

For information on these keywords, see "Definition-Specification Keywords" on page 321. Figure 67 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 67. 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 or PI specification. The following keywords, which apply to defining the type, are allowed on the parameter definition specifications:

ASCEND The array is in ascending sequence.

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

DESCEND The array is in descending sequence.

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 structure 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.

LIKEFILE(filename)
The parameter is a file, either *filename* or a file related through the LIKEFILE keyword to *filename*.

PROCPTR The parameter is a procedure pointer.

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

VARYING{(2|4)}
A character, graphic, or UCS-2 parameter has a variable-length format.

For information on these keywords, see “Definition-Specification Keywords” on page 321.

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 { : opt5 } } } })

Where opt1 ... opt5 can be *NOPASS, *OMIT, *VARSIZE, *STRING, *TRIM, or *RIGHTADJ. For example, **OPTIONS(*VARSIZE : *NOPASS)**.

Specifies the following parameter passing options:

***NOPASS**
The parameter does not have to be passed. If a parameter has OPTIONS(*NOPASS) specified, then all parameters following it must also have OPTIONS(*NOPASS) specified.

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

***VARSIZE**
The parameter may contain less data than is indicated on the definition. This keyword is valid only for character parameters, graphic parameters, UCS-2 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.

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

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

TIP

For the parameter passing options *NOPASS, *OMIT, and *VARSIZE, it is up to the programmer of the procedure to ensure that these options are handled. For example, if OPTIONS(*NOPASS) is coded and you choose **not** to pass the parameter, the procedure must check that the parameter was passed before it accesses it. The compiler will not do any checking for this.

VALUE
The parameter is passed by value.

For information on the keywords listed above, see “Definition-Specification Keywords” on page 321. For more information on using prototyped parameters, see the chapter on calling programs and procedures in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Procedure Interface

If a prototyped program or procedure has call parameters or a return value, then a procedure interface definition must be defined, either in the main source section (for a cycle-main procedure) or in the subprocedure section. If a prototype was specified, the **procedure interface definition** repeats the prototype information within the definition of a procedure. Otherwise, the procedure interface provides the information that allows the compiler to implicitly define the prototype. The procedure interface 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 required for the cycle-main procedure. It is optional for subprocedures. 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 cycle-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 the prototype is explicitly specified.
- If a prototype is not specified, the EXTPGM or EXTPROC keyword may be specified for the procedure interface.

TIP

If a module contains calls to a prototyped program or procedure that is defined in a different module, 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 programs or procedures to other users, be sure to provide them with the prototypes (in /COPY files) as well.

Chapter 8. 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)" on page 711 operation

Note: You can define only run-time arrays in a subprocedure. Tables, prerun-time 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 next section 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. The section after next describes the same information for tables.

Arrays

There are three types of arrays:
- The *run-time array* is loaded by your program while it is running.
- The *compile-time array* is loaded when your program is created. The initial data becomes a permanent part of your program.
- The *prerun-time 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 run-time array. For defining and loading compile-time and prerun-time arrays you 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 (1) the array name, followed by (2) a left parenthesis, followed by (3) an index, followed by (4) 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 you specify 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 your 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 your program.

The Essential Array Specifications

You define an array on a definition specification. Here are the essential specifications for all arrays:

- 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 68 shows an example of the essential array specifications.

Coding a Run-Time Array

If you make no further specifications beyond the essential array specifications, you have defined a *run-time array*. Note that the keywords ALT, CTDATA, EXTFMT, FROMFILE, PERRCD, and TOFILE cannot be used for a run-time array.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
DARC              S              3A   DIM(12)
```

Figure 68. The Essential Array Specifications to Define a Run-Time Array

Loading a Run-Time Array

You can assign initial values for a run-time array using the INZ keyword on the definition specification. You can also assign initial values for a run-time array through input or calculation specifications. This second method may also be used to put data into other types of arrays.

For example, you may 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 may 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 Run-Time Array by Reading One Record from a File

If an input record from a database file will contain all the information for the entire array, the array can be loaded in a single input operation. If the fields in the database record that correspond to the array occupy consecutive positions in the database record, then the array can be loaded with a single Input specification, as shown in Figure 69. The Input specification defines the positions in the database record for the entire array.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
DINPARR          S             12A   DIM(6)
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IARRFILE   AA  01
I                                  1   72  INPARR
```

Figure 69. Using a Run-Time Array with Consecutive Elements

If the fields in the database record that correspond to the array are scattered throughout the database record, then the array must be loaded with a several Input specifications. The example in Figure 70 assumes that the database record contains data for all the array elements, but a blank separates the data for each array element in the database record. Each Input specification defines the position in the database record for a single element.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
DARRX            S             12A   DIM(6)
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC................................
I.......................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
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 70. Defining a Run-Time Array with Scattered Elements

Loading a Run-Time Array by Reading Several Records from A File

If the data for the array is not available in a single record from the database file, the array must be loaded by reading more than one record from the database file. Each record may provide the data for one or more elements of the array. The ILE RPG program processes 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 elements. It may be necessary to suppress calculation and output operations until the entire array is read into the program.

For example, assume that each record from file ARRFILE2 contains the information for one array element in positions 1-12. You can code the Input specification for the array element with a variable index. Your program would set the index before the record was read as shown in Figure 71.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
DARRX             S             12A   DIM(6)
DN                S              5P 0 INZ(1)
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC...............................
I........................Fmt+SPFrom+To+++DcField+++++++++L1M1FrPlMnZr....
IARRFILE2  AA  01
I                                  1   12  ARRX(N)
CL0N01Factor1+++++++Opcode&ExtFactor2;+++++++Result++++++++Len++D+HiLoEq
C                   IF        N = %ELEM(ARR)
 * The array has been loaded
 ..... process the array
 * Set the index to 1 to prepare for the next complete array
C                   EVAL      N = 1
C                   ELSE
 * Increment the index so the next input operation will fill
 * the next array element
C                   EVAL      N = N + 1
C                   ENDIF
```

Figure 71. Loading an array from a file, one element per record

Loading an Array from Identical Externally-Described Fields

If an input record from a externally-described database file has several fields that are defined identically, you can define a data structure that will allow you to process those fields as though they were an array. There are three cases to consider:

1. The fields are consecutive in the record and appear at the beginning of the record.

   ```
   A          R REC
   A            FLD1           5P 0
   A            FLD2           5P 0
   A            FLD3           5P 0
   A            OTHER         10A
   ```

   For this case, you can use an externally-described data structure and define your array as an additional subfield, mapping the array to the fields using the OVERLAY keyword:

   ```
   FMYFILE    IF   E             DISK
   D myDS          E DS                  EXTNAME(MYFILE)
   D   fldArray                          LIKE(FLD1) DIM(3)
   D                                     OVERLAY(myDs)

   C                   READ      MYFILE
   C                   FOR       i = 1 to %ELEM(fldArray)
   C*                     ... process fldArray(i)
   C                   ENDFOR
   ```

2. The fields are consecutive in the record but do not appear at the beginning of the record.

   ```
   A          R REC
   A            OTHER1        10A
   A            ... more fields
   ```

```
     A            FLD1          15A
     A            FLD2          15A
     A            FLD3          15A
     A            OTHER2        10A
```

For this case, you can use an externally-described data structure and define your array as a standalone field, mapping the array to the fields using the BASED keyword, and initializing the basing pointer to the address of the first field.

```
     FMYFILE    IF   E             DISK
     D myDS          E DS                   EXTNAME(MYFILE)

     D fldArray        S                    LIKE(FLD1) DIM(3)
     D                                      BASED(pFldArray)
     D pFldArray       S               *    INZ(%addr(FLD1))

     C                   READ      MYFILE
     C                   FOR       i = 1 to %ELEM(fldArray)
     C*                     ... process fldArray(i)
     C                   ENDFOR
```

3. The fields are not consecutive in the record.

```
     A          R REC
     A            OTHER1        10A
     A            FLD1            T         TIMFMT(*ISO)
     A            FLD2            T         TIMFMT(*ISO)
     A            OTHER2        10A
     A            FLD3            T         TIMFMT(*ISO)
     A            OTHER3        10A
```

For this case, you must define a program-described data structure and list the fields to be used for the array without defining any type information. Then map the array to the fields using the OVERLAY keyword.

```
     FMYFILE    IF   E             DISK
     D myDS            DS
     D   FLD1
     D   FLD2
     D   FLD3
     D   fldArray                           LIKE(FLD1) DIM(3)
     D                                      OVERLAY(myDs)

     C                   READ      MYFILE
     C                   FOR       i = 1 to %ELEM(fldArray)
     C*                     ... process fldArray(i)
     C                   ENDFOR
```

Sequencing Run-Time Arrays

Run-time 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 plus the keyword CTDATA. In addition, on a definition specification you can specify:

- The number of array entries in an input record using the PERRCD keyword. If the keyword is not specified, the number of entries defaults to 1.

- The external data format using the EXTFMT keyword. The only allowed values are L (left-sign), R (right-sign), or S (zoned-decimal). The EXTFMT keyword is not allowed for float compile-time arrays.
- A file to which the array is to be written when the program ends with LR on. You specify this using the TOFILE keyword.

See Figure 72 for an example of a compile-time array.

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 **ALTSEQ, **CTDATA, and **FTRANS keywords, the array data may be entered in anywhere following the source records. If you do not use those keywords, the array data must follow the source records, and any alternate collating sequence or file translation 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 72).
- 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 72. 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; therefore, 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:
  1. **CTDATA arrayname: The data for the array may be specified anywhere in the compile-time data section.
  2. **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).
- For ascending or descending character arrays when ALTSEQ(*EXT) is specified on the control specification, the alternate collating sequence is used for the sequence checking. If the actual collating sequence is not known at compile time (for example, if SRTSEQ(*JOBRUN) is specified on a control specification or as a command parameter) the alternate collating sequence table will be retrieved at runtime and the checking will occur during initialization at *INIT. Otherwise, the checking will be done at compile time.
- Graphic and UCS-2 arrays will be sorted by hexadecimal values, regardless of the alternate collating sequence.
- If L or R is specified on the EXTFMT keyword on the definition specification, each element must include the sign (+ or -). An array with an element size of 2 with L specified would require 3 positions in the source data as shown in the following example.

```
*....+....1....+....2....+....3....+....4....+....5....+....6....+....*
      DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++
      D UPDATES                         2 0  DIM(5) PERRCD(5) EXTFMT(L) CTDATA
**CTDATA UDPATES
+37-38+52-63-49+51
```

- 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.
- Graphic data must be enclosed in shift-out and shift-in characters. If several elements of graphic data are included in a single record (without intervening nongraphic data) only one set of shift-out and shift-in characters is required for the record. If a graphic array is defined in alternating format with a nongraphic array, the shift-in and shift-out characters must surround the graphic data. If two graphic arrays are defined in alternating format, only one set of shift-in and shift-out characters is required for each record.

Coding a Prerun-Time Array

In addition to the essential array specifications, you can also code the following specifications or keywords for prerun-time arrays.

On the definition specifications, you can specify

- The name of the file with the array input data, using the FROMFILE keyword.
- The name of a file to which the array is written at the end of the program, using the TOFILE keyword.
- The number of elements per input record, using the PERRCD keyword.
- The external format of numeric array data using the EXTFMT keyword.
- An alternating format using the ALT keyword.

Note: The integer or unsigned format cannot be specified for arrays defined with more than ten digits.

On the file-description specifications, you can specify a T in position 18 for the file with the array input data.

Example of Coding Arrays

Figure 73 shows the definition specifications required for two prerun-time arrays, a compile-time array, and a run-time array.

```
*....+....1....+....2....+....3....+....4....+....5....+....6....+....*
      HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      H DATFMT(*USA) TIMFMT(*HMS)
      D*ame+++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++
       * Run-time array.  ARI has 10 elements of type date. They are
       * initialized to September 15, 1994.  This is in month, day,
       * year format using a slash as a separator as defined on the
       * control specification.
      DARI              S                D   DIM(10) INZ(D'09/15/1994')
       *
       * Compile-time arrays in alternating format.  Both arrays have
       * eight elements (three elements per record).  ARC is a character
       * array of length 15, and ARD is a time array with a predefined
       * length of 8.
      DARC              S              15    DIM(8) PERRCD(3)
      D                                      CTDATA
      DARD              S                T   DIM(8) ALT(ARC)
       *
       * Prerun-time array.  ARE, which is to be read from file DISKIN,
       * has 250 character elements (12 elements per record).  Each
       * element is five positions long.  The size of each record
       * is 60 (5*12). The elements are arranged in ascending sequence.
      DARE              S               5A   DIM(250) PERRCD(12) ASCEND
      D                                      FROMFILE(DISKIN)
       *
       * Prerun-time array specified as a combined file.  ARH is written
       * back to the same file from which it is read when the program
       * ends normally with LR on.  ARH has 250  character elements
       * (12 elements per record).  Each elements is five positions long.
       * 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 73. Definition Specifications for Different Types of Arrays

Loading a Prerun-Time Array

For a *prerun-time array*, enter array input data into a file. The file must be a sequential program described file. During initialization, 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 prerun-time 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 164.

Sequence Checking for Character Arrays

Sequence checking for character arrays that have not been defined with ALTSEQ(*NONE) has two dependencies:

1. Whether the ALTSEQ control specification keyword has been specified, and if so, how.
2. Whether the array is compile time or prerun time.

The following table indicates when sequence checking occurs.

Control Specification Entry	ALTSEQ Used for SORTA, LOOKUP and Sequence Checking	When Sequence Checked for Compile Time Array	When Sequence Checked for Prerun Time Array
ALTSEQ(*NONE)	No	Compile time	Run time
ALTSEQ(*SRC)	No	Compile time	Run time
ALTSEQ(*EXT) (known at compile time)	Yes	Compile time	Run time
ALTSEQ(*EXT) (known only at run time)	Yes	Run time	Run time

Note: For compatibility with RPG III, SORTA and LOOKUP do not use the alternate collating sequence with ALTSEQ(*SRC). If you want these operations to be performed using the alternate collating sequence, you can define a table on the system (object type *TBL), containing your alternate sequence. Then you can change ALTSEQ(*SRC) to ALTSEQ(*EXT) on your control specification and specify the name of your table on the SRTSEQ keyword or parameter of the create command.

Initializing Arrays

Run-Time Arrays

To initialize each element in a run-time 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 Prerun-Time Arrays

The INZ keyword cannot be specified for a compile-time or prerun-time 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 prerun-time 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 prerun-time 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-run time arrays are initialized after compile-time arrays.

If a subfield initialization overlaps a compile-time or prerun-time array, the initialization of the array takes precedence; that is, 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 prerun-time 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 74 shows two arrays, ARRA and ARRB, in alternating format.

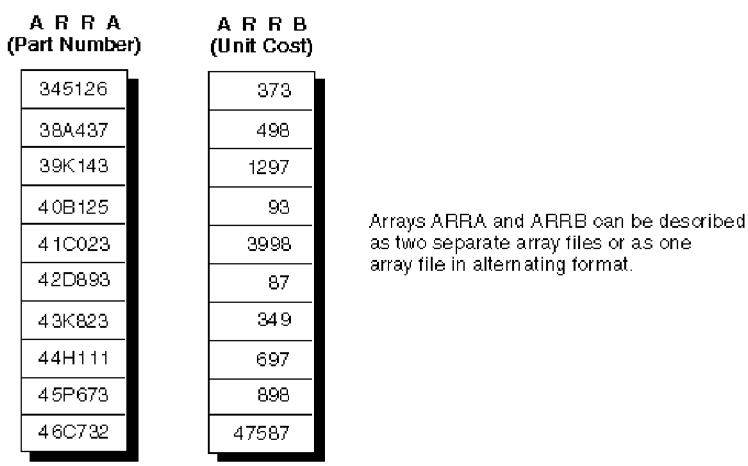


Figure 74. Arrays in Alternating and Nonalternating Format

The records for ARRA and ARRB look like the records below when described as two separate array files.

This record contains ARRA entries in positions 1 through 60.

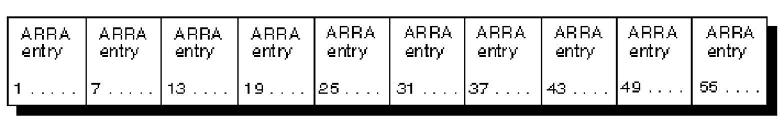


Figure 75. Arrays Records for Two Separate Array Files

This record contains ARRB entries in positions 1 through 50.

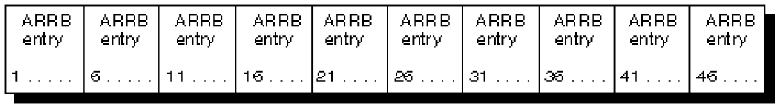


Figure 76. Arrays Records for One Array File

The records for ARRA and ARRB look like the records below 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.

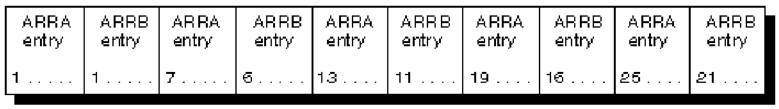


Figure 77. Arrays Records for One Array File in Alternating Format

```
      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
```

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 171
- "Searching an Array Without an Index" on page 170
- "LOOKUP (Look Up a Table or Array Element)" on page 711

For more information about the %LOOKUPxx built-in functions, see "%LOOKUPxx (Look Up an Array Element)" on page 551.

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 requested.

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 78 shows an example of a LOOKUP on an array without an index.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
FARRFILE   IT   F    5        DISK
F*
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
DDPTNOS           S             5S 0 DIM(50) FROMFILE(ARRFILE)
D*
CL0N01Factor1+++++++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 78. 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 Data Structure

You can use the %LOOKUP built-in function to search an array data structure using one of its subfields as a key.

For more information about searching an array data structure, see "%LOOKUPxx (Look Up an Array Element)" on page 551.

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 79 shows a LOOKUP on an array with an index.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
FFilename++IPEASFRlen+LKlen+AIDevice+.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*
CL0N01Factor1+++++++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 not found that is equal to the search argument,
C* element X of DPTDSC is moved to DPTNAM.
C                   IF        NOT *IN20
C                   MOVE      DPTDSC(X)     DPTNAM           20
C                   ENDIF
```

Figure 79. LOOKUP Operation on an Array with an Index

This example shows the same array of department numbers, DPTNOS, as Figure 78 on page 170. 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.

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. You can process individual elements 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, Z-ADD, SUB, Z-SUB, MULT, DIV, SQRT, ADDDUR, SUBDUR, EVAL, EXTRCT, MOVE, MOVEL, MOVEA, MLLZO, MLHZO, MHLZO, MHHZO, DEBUG, XFOOT, LOOKUP, SORTA, PARM, DEFINE, CLEAR, RESET, CHECK, CHECKR, and SCAN.

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, COMP, CABxx, TESTZ, TESTN, TESTB, MVR, DO, DOUxx, DOWxx, DOU, DOW, IFxx, WHENxx, WHEN, IF, SUBST, and CAT.

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, Z-ADD, EVAL, SUB, Z-SUB, ADDDUR, SUBDUR, EXTRCT, MULT, DIV, SQRT, MOVE, MOVEL, MLLZO, MLHZO, MHLZO and MHHZO. The following rules apply to these operations when an array name without an index is specified:

- When factors 1 and 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.
- In an EVAL expression, if any arrays on the right-hand side are specified without an index, the left-hand side must also contain an array without an index.

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 494 for more detail.

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 584 for more detail.

Sorting Arrays

You can sort an array or a section of an array using the "SORTA (Sort an Array)" on page 815 operation code. The array is sorted into sequence (ascending or descending), depending on the sequence specified for the array on the definition specification. If no sequence is specified for the array, the sequence defaults to ascending sequence, but you can sort in descending sequence by specifying the 'D' operation extender.

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
CL0N01Factor1+++++++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 80. SORTA Operation with OVERLAY

Sorting an Array Data Structure

You can use the SORTA operation to sort an array data structure using one of its subfields as a key.

For more information about sorting an array data structure, see "SORTA (Sort an Array)" on page 815.

Array Output

Entire arrays can be written out under ILE RPG control only at end of program when the LR indicator is 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 output or 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 chapter).

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: that is, 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.

Editing of entire arrays is only valid in output specifications, not with the %EDITC or %EDITW built-in functions.

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 built-in 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 built-in 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 81. 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.

Loading
Tables can be loaded only at compilation time and prerun-time.

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. An index cannot be specified for a table.

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"
- "LOOKUP (Look Up a Table or Array Element)" on page 711

For more information about the %TLOOKUPxx built-in functions, see "%TLOOKUPxx (Look Up a Table Element)" on page 593.

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), that 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...
CL0N01Factor1+++++++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 82. 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 83).

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++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 83. Specifying the Table Element Found in LOOKUP Operations

Chapter 9. Data Types and Data Formats

This chapter describes the data types supported by RPG IV and their special characteristics. The supported data types are:

- Character Format
- Numeric Data Type
- Graphic Format
- UCS-2 Format
- Date Data Type
- Time Data Type
- Timestamp Data Type
- Object Data Type
- Basing Pointer Data Type
- Procedure Pointer Data Type

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

Internal and External Formats

Numeric, character, date, time, 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 run-time performance of arithmetic operations is important. For more information, see “Performance Considerations” on page 435.

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.

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 435.

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

Table 31. 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

Specifying an External Format for a Numeric Field

For any of the fields in Table 31, 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 31 on page 180, 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.

Character Data Type

The character data type represents character values and may have any 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. 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	If CCSID(*CHAR:*JOBRUN) is specified on the Control specification, the character CCSID is assumed to be the runtime job CCSID. Otherwise, the CCSID is assumed to be the mixed graphic CCSID related to the job CCSID.
Indicator	One **single-byte** character that is fixed in length	If CCSID(*CHAR:*JOBRUN) is specified on the Control specification, the character CCSID is assumed to be the runtime job CCSID. Otherwise, the CCSID is assumed to be the mixed graphic CCSID related to the job CCSID.
Graphic	One or more **double-byte** characters that are fixed or variable in length	65535 or a CCSID with the EBCDIC double-byte encoding scheme (x'1200')
UCS-2	One or more **double-byte** characters that are fixed or variable in length	13488 or a CCSID with the UCS-2 encoding scheme (X'7200')

For information on the CCSIDs of character data, see “Conversion between Character, Graphic and UCS-2 Data” on page 194.

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 Formats” on page 185.

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.

A special set of predefined RPG IV indicators (*INxx) is also available. For a description of these indicators, see Chapter 4, “RPG IV Indicators,” on page 47.

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 are implicitly defined with ALTSEQ(*NONE). This means that the alternate collating sequence is not used for comparisons involving indicators.
- Indicators may be used as key-fields 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.

Fields defined as graphic data do not contain shift-out (SO) or shift-in (SI) characters. The difference between single byte character and double byte graphic data is shown in the following figure:

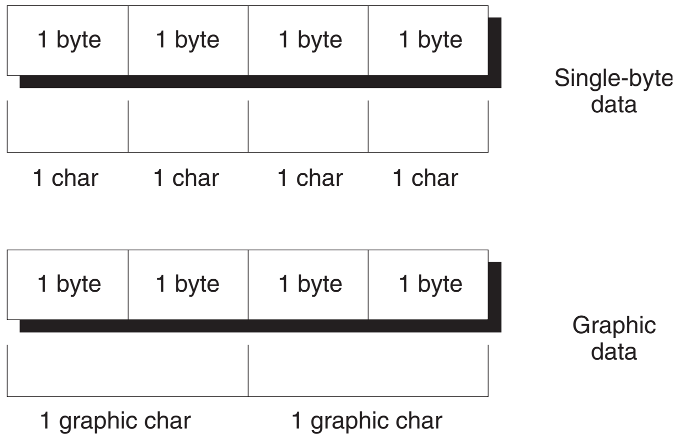


Figure 84. Comparing Single-byte and graphic data

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

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 graphic format, see "Variable-Length Character, Graphic and UCS-2 Formats" on page 185.

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 X'4040'. The value of *HIVAL is X'FFFF', and the value of *LOVAL is X'0000'.

Note: The examples of graphic literals in this manual are not valid graphic literals. They use the letter 'o' to represent the shift-out character and the letter 'i' to represent the shift-in character. Often the graphic data is expressed as D1D2 or AABB; these are not valid double-byte characters. Normally, graphic literals are entered using a DBCS-capable keyboard that automatically enters the shift-out and shift-in characters before and after the DBCS characters are entered.

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.

Fields defined as UCS-2 data do not contain shift-out (SO) or shift-in (SI) characters.

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 Formats."

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'. You can specify the initialization value for UCS-2 fields using character, UCS-2 or Graphic values. If the type of the literal is not UCS-2, the compiler will perform an implicit conversion to UCS-2. For example, to initialize a UCS-2 field with the UCS-2 form of 'abc', you can specify INZ('abc'), INZ(%UCS2('abc')) or INZ(U'006100620063').

For more information on the UCS-2 format, see the iSeries Information Center globalization topic.

Variable-Length Character, Graphic and UCS-2 Formats

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 or 4 bytes longer than the declared maximum length, depending on how the VARYING keyword is specified for the field. The leftmost 2 or 4 bytes are an unsigned integer field containing the current length in characters, graphic characters or UCS-2 characters. The actual data starts at the third or fifth byte of the variable-length field. Figure 85 shows how variable-length character fields are stored:

```
------------------------------------
    | current  | character data |
    | length   |                |
------------------------------------
     UNS(V)       CHAR(N)

 N = declared maximum length
 V = number of bytes specified for the length prefix

 V + N = total number of bytes
```

Figure 85. Character Fields with Variable-Length Format

The unsigned integer length prefix can be either two bytes long or four bytes long. You indicate the size of the prefix using the parameter of the VARYING keyword, either VARYING(2) or VARYING(4). If you specify VARYING without a parameter, a size of 2 is assumed if the specified length is between 1 and 65535; otherwise, a size of 4 is assumed.

Figure 86 on page 186 shows how variable-length graphic fields are stored. UCS-2 fields are stored similarly.

```
------------------------------------
   | current  | graphic-data    |
   | length   |                 |
------------------------------------
     UNS(V)       CHAR(N)

 N = declared maximum length = number of double bytes
 V = number of bytes specified for the length prefix

 V + 2(N) = total number of bytes
```

Figure 86. 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 data 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.

A variable-length field is initialized by default to have a current length of zero.

You can obtain the address of the data portion of a variable-length field using %ADDR(fieldname:*DATA).

For examples of using variable-length fields, see:
- "Using Variable-Length Fields" on page 189
- "%LEN (Get or Set Length)" on page 547
- "%CHAR (Convert to Character Data)" on page 505
- "%REPLACE (Replace Character String)" on page 568
- "%ADDR (Get Address of Variable)" on page 494

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 16773100 single-byte characters and from 1 to 8386550 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.
- Variable-length fields which have different-sized length prefixes are fully compatible except when passed as reference parameters.
- When a prototyped parameter is defined with the VARYING keyword, and without either the CONST or VALUE keyword, the passed parameters must have the same size of length prefix as the prototyped parameter. This rule applies even if OPTIONS(*VARSIZE) is specified.
- In all cases except subfields defined using positional notation, the length (specified by the LEN keyword or the length entry in positions 33-39 on the definition specifications) contains the maximum length of the field in characters; this length does not include the 2- or 4-byte length prefix.

- For subfields defined using positional notation, the size specified by the From and To positions includes the 2- or 4-byte length prefix. As a result, the number of bytes that you specify using the positional notation must be two or four bytes longer than the number of bytes required to hold the data. If you specify VARYING(2), you add two bytes to the bytes required for the data; if you specify VARYING(4), you add four bytes. If you specify VARYING without a parameter, you add two bytes if the length is 65535 or less, and you add four bytes if the length is greater than 65535. For alphanumeric subfields, sizes from 3 to 65537 represent lengths of 1 to 65535; for UCS-2 and Graphic subfields, sizes from 5 to 131072 represent lengths of 1 to 65535.

  Note: A more convenient way to specify variable-length subfields is to use length notation, and to use the OVERLAY keyword to specify the position of the subfield within the 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 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.
- *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   CONST VARYING OPTIONS(*VARSIZE)
D   FromStr                  32765A   CONST VARYING OPTIONS(*VARSIZE)
D   ToStr                    32765A   CONST VARYING OPTIONS(*VARSIZE)
D   StartPos                     5U 0 VALUE
D   Replaced                     5U 0 OPTIONS(*OMIT)
```

Figure 87. 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 88. 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.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
C                   EVAL      fld = 'XY'
```

The DSPLY operation changes the length of a variable-length result field to the length of the value entered by the user. For example, if the result field is a character field of length 10, and the value entered by the user is '12345', the length of the field will be set to 5 by the DSPLY operation.

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'
CL0N01Factor1+++++++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 built-in 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'
CL0N01Factor1+++++++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 or use CEEDOD within your subrocedure to get the actual length of the parameter.

CVTOPT(*VARCHAR) and CVTOPT(*VARGRAPHIC)

The ILE RPG compiler can internally define variable-length character, graphic, or UCS-2 fields from an externally described file or data structure as fixed-length character fields. Although converting variable-length character, graphic, and UCS-2 fields to fixed-length format is not necessary, CVTOPT 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 or UCS-2 fields) on the CVTOPT control specification keyword or command parameter. 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 ILE 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 in an ILE RPG program as a fixed-length character field.
- For single-byte character fields, the length of the declared ILE RPG field is the length of the DDS field plus 2 bytes.
- For DBCS-graphic data fields, the length of the declared ILE RPG field is twice the length of the DDS field plus 2 bytes.
- The two extra bytes in the ILE RPG field contain a unsigned integer number which represents the current length of the variable-length field. Figure 89 shows the ILE RPG 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.

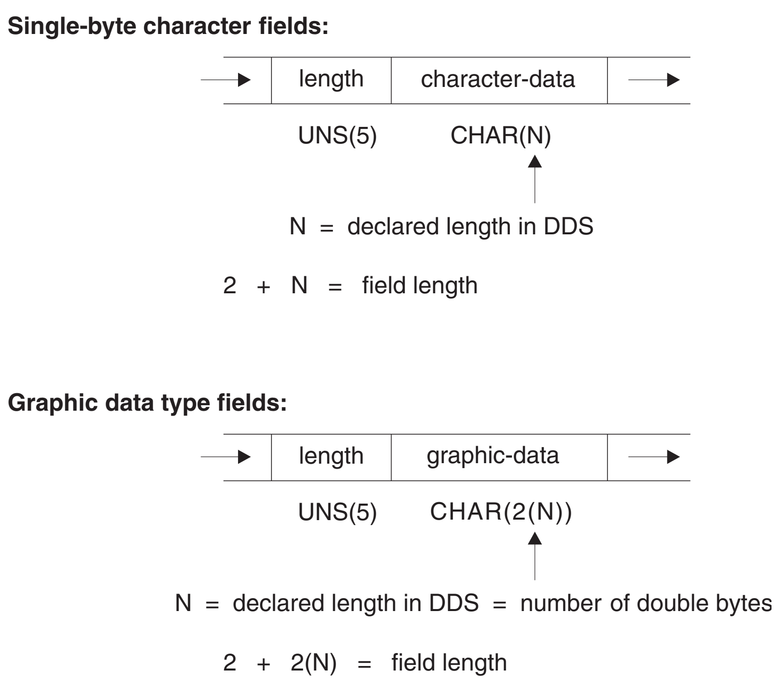


Figure 89. ILE RPG Field Length of Converted Variable-Length Fields

- Your ILE RPG 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 will occur for an output operation if the first two bytes of the field contain invalid field-length data.
- Control-level indicators, match field entries, and field indicators are not allowed on an input specification if the input field is a variable-length field from an externally described input file.

- Sequential-within-limits processing is not allowed when a file contains variable-length key fields.
- 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 either not specified on the output specification or is ignored in the ILE RPG program, the ILE RPG compiler will place a default value 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 an ILE RPG 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....+.. *
 *
 *   Specify the CVTOPT(*VARCHAR) keyword on a control
 *   specification or compile the ILE RPG program with
 *   CVTOPT(*VARCHAR) on the command.
 *
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *
H CVTOPT(*VARCHAR)
 *
 *   Externally described file name is MASTER.
 *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
 *
FMASTER    UF   E             DISK
 *
 *   FLDVAR is a variable-length field defined in DDS with
 *   a DDS length of 100.  Notice that the RPG field length
 *   is 102.
 *
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
 *
D                 DS
D FLDVAR                 1    102
D   FLDLEN                      5U 0 OVERLAY(FLDVAR:1)
D   FLDCHR                    100    OVERLAY(FLDVAR:3)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 * A character value is moved to the field FLDCHR.
 * After the CHECKR operation, FLDLEN has a value of 5.
C                   READ      MASTER                                 LR
C                   MOVEL     'SALES'       FLDCHR
C     ' '           CHECKR    FLDCHR        FLDLEN
C  NLR              UPDATE    REC
```

Figure 90. Converting a Variable-Length Character Field

If you would like to use a converted variable-length graphic field, 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.

```
 *
 *   Specify the CVTOPT(*VARGRAPHIC) keyword on a control
 *   specification or compile the ILE RPG program with
 *   CVTOPT(*VARGRAPHIC) on the command.
 *
 *   The variable-length graphic field VGRAPH is declared in the
 *   DDS as length 3.  This means the maximum length of the field
 *   is 3 double bytes, or 6 bytes.  The total length of the field,
 *   counting the length portion, is 8 bytes.
 *
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
 *
D                 DS
DVGRAPH                         8
D  VLEN                         4U 0 OVERLAY(VGRAPH:1)
D  VDATA                        3G   OVERLAY(VGRAPH:3)
 *
 *   Assume GRPH is a fixed-length graphic field of length 2
 *   double bytes.  Copy GRPH into VGRAPH and set the length of
 *   VGRAPH to 2.
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C                   MOVEL     GRPH          VDATA
C                   Z-ADD     2             VLEN
```

Figure 91. Converting a Variable-Length Graphic Field

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 or between character and graphic data (unless graphic CCSIDs are being ignored).

When converting between character and graphic data, the CCSID of the character data is assumed to be the graphic CCSID related to the job CCSID.

When converting between character and UCS-2 data, if CCSID(*CHAR:*JOBRUN) is specified on the control specification, the CCSID of the character data is assumed to be job CCSID. Otherwise, it is assumed to be the mixed-byte CCSID related to the job CCSID.

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 or 1200.

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.

Conversions

Conversion between character and double-byte graphic fields consists of adding or removing shift-out and shift-in bracketing and possibly performing CCSID conversion on the graphic data.

When you use character, graphic, and UCS-2 values with different types or CCSIDs in the same operation, conversions must be done to ensure that all the values have the same type and CCSID. The conversions can be done explicitly, using the conversion built-in functions %CHAR, %UCS2 or %GRAPH. However, in the following scenarios, the conversion built-in functions do not have to be specified; the compiler will do the conversions implicitly when necessary:

Comparison
: Both operands are converted to UCS-2 before comparison.

Assignment
: The source value is converted to the type and CCSID of the target value.

Parameters passed by value and by read-only reference
: The passed parameter is converted to the type and CCSID of the prototyped parameter.

Note: While implicit conversion is supported for the result of a concatenation expression, all the operands of the concatenation expression must have the same type and CCSID.

Alternate Collating Sequence

The alternate collating sequence applies only to single-byte character data.

Each character is represented internally by a hexadecimal value, which governs the order (ascending or descending sequence) of the characters and is known as the normal collating sequence. The alternate collating sequence function can be used to alter the normal collating sequence. This function also can be used to allow two or more characters to be considered equal.

Changing the Collating Sequence

Using an alternate collating sequence means modifying the collating sequence for character match fields (file selection) and character comparisons. You specify that an alternate collating sequence will be used by specifying the ALTSEQ keyword on the control specification. The calculation operations affected by the alternate collating sequence are ANDxx, COMP, CABxx, CASxx, DOU, DOUxx, DOW, DOWxx, IF, IFxx, ORxx, WHEN, and WHENxx. This does not apply to graphic or UCS-2 compare operations. LOOKUP and SORTA are affected only if you specify ALTSEQ(*EXT). The characters are not permanently changed by the alternate collating sequence, but are temporarily altered until the matching field or character compare operation is completed.

Use the ALTSEQ(*NONE) keyword on the definition specification for a variable to indicate that when the variable is being compared with other character data, the normal collating sequence should always be used even if an alternate collating sequence was defined.

Changing the collating sequence does not affect the LOOKUP and SORTA operations (unless you specify ALTSEQ(*EXT)) or the hexadecimal values assigned to the figurative constants *HIVAL and *LOVAL. However, changing the collating sequence can affect the order of the values of *HIVAL and *LOVAL in the collating sequence. Therefore, if you specify an alternate collating sequence in your program and thereby cause a change in the order of the values of *HIVAL and *LOVAL, undesirable results may occur.

Using an External Collating Sequence

To specify that the values in the SRTSEQ and LANGID command parameters or control specification keywords should be used to determine the alternate collating sequence, specify ALTSEQ(*EXT) on the control specification. For example, if ALTSEQ(*EXT) is used, and SRTSEQ(*LANGIDSHR) and LANGID(*JOBRUN) are specified, then when the program is run, the shared-weight table for the user running the program will be used as the alternate collating sequence.

Since the LOOKUP and SORTA operations are affected by the alternate collating sequence when ALTSEQ(*EXT) is specified, character compile-time arrays and tables are sequence-checked using the alternate collating sequence. If the actual collating sequence is not known until runtime, the array and table sequence cannot be checked until runtime. This means that you could get a runtime error saying that a compile-time array or table is out of sequence.

Pre-run arrays and tables are also sequence-checked using the alternate collating sequence when ALTSEQ(*EXT) is specified.

Note: The preceding discussion does not apply for any arrays and tables defined with ALTSEQ(*NONE) on the definition specification.

Specifying an Alternate Collating Sequence in Your Source

To specify that an alternate collating sequence is to be used, use the ALTSEQ(*SRC) keyword on the control specification. If you use the **ALTSEQ, **CTDATA, and **FTRANS keywords in the compile-time data section, the alternate-collating sequence data may be entered anywhere following the source records. If you do not use those keywords, the sequence data must follow the source records, and the file translation records but precede any compile-time array data.

If a character is to be inserted between two consecutive characters, you must specify every character that is altered by this insertion. For example, if the dollar sign ($) is to be inserted between A and B, specify the changes for character B onward.

See Appendix B, “EBCDIC Collating Sequence,” on page 909 for the EBCDIC character set.

Formatting the Alternate Collating Sequence Records

The changes to the collating sequence must be transcribed into the correct record format so that they can be entered into the system. The alternate collating sequence must be formatted as follows:

Record Position	Entry
1-6	ALTSEQ (This indicates to the system that the normal sequence is being altered.)
7-10	Leave these positions blank.
11-12	Enter the hexadecimal value for the character whose normal sequence is being changed.
13-14	Enter the hexadecimal value of the character replacing the character whose normal sequence is being changed.
15-18 19-22 23-26 ... 77-80	All groups of four beginning with position 15 are used in the same manner as positions 11 through 14. In the first two positions of a group enter the hexadecimal value of the character to be replaced. In the last two positions enter the hexadecimal value of the character that replaces it.

The records that describe the alternate collating sequence must be preceded by a record with **ƀ (ƀ = blank) in positions 1 through 3. The remaining positions in this record can be used for comments.

```
     HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    H ALTSEQ(*SRC)
    DFLD1           s             4A   INZ('abcd')
    DFLD2           s             4A   INZ('ABCD')
**
ALTSEQ    81C182C283C384C4
```

Numeric Data Type

The numeric data type represents numeric values. Numeric data has one of the following formats:

B Binary Format

F Float Format

I Integer Format

P Packed-Decimal Format

U Unsigned Format

Z Zoned-Decimal Format

The default initialization value for numeric fields is zero.

Binary Format

Binary format means that the sign (positive or negative) is in the leftmost bit of the field and the numeric 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. 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.

Processing of a Program-Described Binary Input Field

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.

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.

Processing of an Externally Described Binary Input Field

The number of digits of a binary 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 RPG IVcompiler handles the data like this:

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 RPG IV program.

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 stand-alone field.

Note that an externally described binary field may have a value outside of the range allowed by RPG IV 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.

Float Format

The float format consists of two parts:

- the mantissa and
- 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, etc.

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 floating-point field are:

4-byte float (8 digits)
: -3.4028235E+38 to -1.1754944E-38, 0.0E+0, +1.1754944E-38 to +3.4028235E+38

8-byte float (16 digits)
: -1.797693134862315E+308 to -2.225073858507201E-308, 0.0E+0, +2.225073858507201E-308 to +1.797693134862315E+308

Note: Float variables conform to the IEEE standard as supported by the IBM i operating system. 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 180 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.
- Output of float values on a dump listing.
- 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 323.
- Length adjustment is not allowed when the LIKE keyword is used to define a field like a float field.
- Float input fields cannot be defined as match or control fields.

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; 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 323.
- 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.
- Integer input fields cannot be defined as match or control fields.

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. The standard signs are used: hexadecimal F for positive numbers and hexadecimal D for negative numbers. The packed-decimal format looks like this:

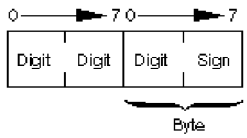


Figure 92. Packed-Decimal Format

The sign portion of the low-order byte indicates whether the numeric value represented in the digit portions is positive or negative. Figure 94 on page 205 shows what the decimal number 21544 looks like in packed-decimal format.

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 63.

Packed fields can be up to 32 bytes long. Table 32 shows the packed equivalents for zoned-decimal fields up to 63 digits long:

Table 32. Packed Equivalents for Zoned-Decimal Fields up to 63 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, 31	16
⋮	⋮
60, 61	31
62, 63	32

For example, an input field read in packed-decimal format has a length of five bytes (as specified on the input or definition 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" on page 361 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 200.

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:

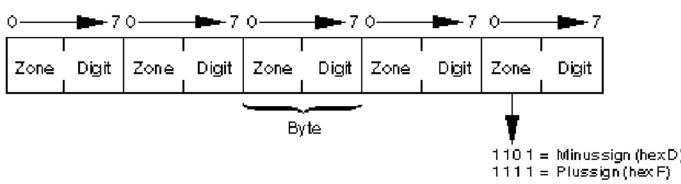


Figure 93. Zoned-Decimal Format

The zone portion of the low-order byte indicates the sign (positive or negative) of the decimal number. The standard signs are used: hexadecimal F for positive numbers and hexadecimal D for negative numbers. In zoned-decimal format, each digit in a decimal number includes a zone portion; however, only the low-order zone portion serves as the sign. Figure 94 on page 205 shows what the number 21544 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.)

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 4E, and a minus sign is a hexadecimal 60.

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 'F'. 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 X'F6F2D5' which appears as 62N.

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.
- 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

Figure 94 on page 205 shows what the decimal number 21544 looks like in various formats.

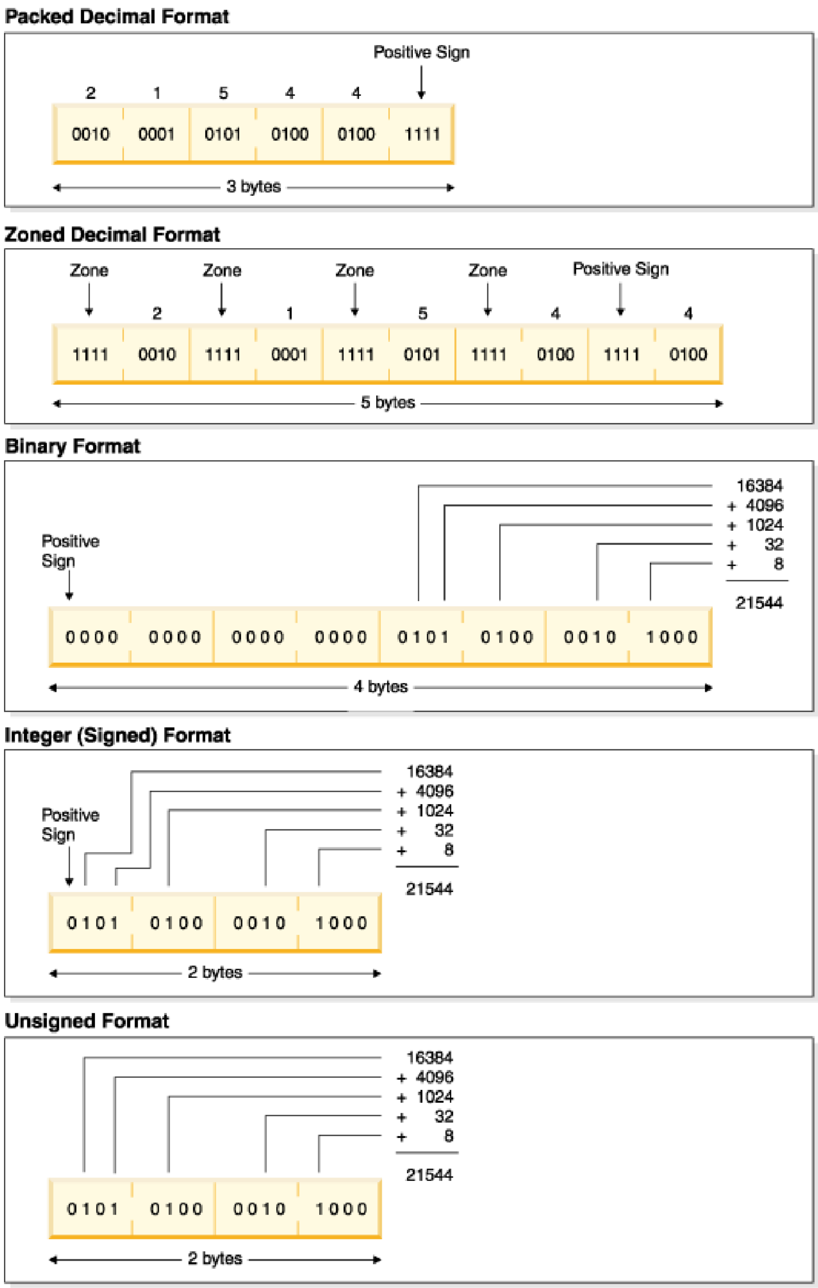


Figure 94. Representation of the Number 21544 in each of the Numeric Formats

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 95 on page 206 shows the number -21544 in integer format.

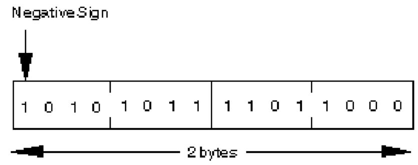


Figure 95. Integer Representation of the Number -21544

Date Data Type

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. Also, dates used for I/O operations such as input fields, output fields or key fields are also 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 33 on page 207 lists the RPG-defined formats for date data and their separators.

For examples on how to code date fields, see the examples in:

- “Date Operations” on page 449
- “Moving Date-Time Data” on page 462
- “ADDDUR (Add Duration)” on page 610
- “MOVE (Move)” on page 720
- “EXTRCT (Extract Date/Time/Timestamp)” on page 689
- “SUBDUR (Subtract Duration)” on page 822
- “TEST (Test Date/Time/Timestamp)” on page 829

Table 33. 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

Table 34 lists the *LOVAL, *HIVAL, and default values for all the RPG-defined date formats.

Table 34. 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

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. It also applies to the 2-digit year formats when *JOBRUN is specified.

*JOBRUN should be used when the field which it is describing is known to have the attributes from the job. For instance, a 12-digit numeric result of a TIME operation will be in the job date format.

Table 35 on page 208 lists the valid externally defined date formats that can be used in Factor 1 of a MOVE, MOVEL, and TEST operation.

Table 35. Externally defined date formats and separators

Format Name	Description	Format (Default Separator)	Valid Separators	Length	Example
2-Digit Year Formats					
*JOBRUN[1]	Determined at runtime from the DATFMT, or DATSEP job values.				
3-Digit Year Formats[2]					
*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

Notes:

1. *JOBRUN is valid only for character or numeric dates with a 2-digit year since the run-time job attribute for DATFMT can only be *MDY, *YMD, *DMY or *JUL.
2. Valid values for the century character 'c' are:

```
 'c'          Years
-----------------------
  0         1900-1999
  1         2000-2099
  .             .
  .             .
  .             .
  9         2800-2899
```

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 720, “MOVEL (Move Left)” on page 741 and “TEST (Test Date/Time/Timestamp)” on page 829.

Initialization

To initialize the Date field to the system date at runtime, specify INZ(*SYS) on the definition specification. To initialize the Date field to the job date at runtime, specify INZ(*JOB) on the definition specification. *SYS or *JOB cannot be used with a field that is exported. The Date field can also be initialized to a literal, named constant or figurative constant.

Note: Runtime initialization takes place after static intitialization.

Time Data Type

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. Also, times used for I/O operations such as input fields, output fields or key fields are also 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 449
- "Moving Date-Time Data" on page 462
- "ADDDUR (Add Duration)" on page 610
- "MOVE (Move)" on page 720
- "SUBDUR (Subtract Duration)" on page 822
- "TEST (Test Date/Time/Timestamp)" on page 829

Table 36 shows the time formats supported and their separators.

Table 36. Time formats and separators for Time data type

RPG Format 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

Table 37 lists the *LOVAL, *HIVAL, and default values for all the time formats.

Table 37. Time Values

RPG 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 720.

Initialization

To initialize the Time field to the system time at runtime, specify INZ(*SYS) on the definition specification. *SYS cannot be used with a field that is exported. The Time field can also be initialized at runtime to a literal, named constant or figurative constant.

Note: Runtime initialization takes place after static intitialization.

*JOBRUN

A special value of *JOBRUN can be used in Factor 1 of a MOVE, MOVEL or TEST operation. This indicates that the separator of the field being described is based on the run-time job attributes, TIMSEP.

Timestamp Data Type

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 zeros. 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. The *LOVAL value for timestamp is 0001-01-01-00.00.00.000000.

For examples on how to code timestamp fields, see the examples in

- “Date Operations” on page 449
- “Moving Date-Time Data” on page 462
- “ADDDUR (Add Duration)” on page 610
- “MOVE (Move)” on page 720
- “SUBDUR (Subtract Duration)” on page 822

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 829.

Initialization

To initialize the Timestamp field to the system date at runtime, specify INZ(*SYS) on the definition specification. *SYS cannot be used with a field that is exported. The Timestamp field can also be initialized at runtime to a literal, named constant or figurative constant.

Note: Runtime initialization takes place after static intitialization.

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 96. Object Data Type Example

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:
If we set pointer PTR1 to point to the G,

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

```
   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': If MY_FIELD is now changed by an EVAL statement to 'HELLO', the storage

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

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 325) 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 16 bytes long and must be aligned on a 16 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 140.

The default initialization value for basing pointers is *NULL.

Note: 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 101 on page 218 shows some examples of how *not* to code basing pointers.

Note: 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.

Note: When a data structure contains a pointer, and the data structure is copied to a character field, or to another data structure that does not have a pointer subfield defined, the pointer information may be lost in the copied value. The actual 16-byte value of the pointer will be copied, but there is extra information in the system that indicates that the 16-byte area contains a pointer; that extra information may not be set in the copied value.

If the copied value is copied back to the original value, the pointer may be lost in the original value.

Passing a data structure containing pointers as a prototyped parameter by read-only reference (CONST keyword) or by value (VALUE keyword) may lose pointer information in the received parameter, if the parameter is prototyped as a character value rather than using the LIKEDS keyword. A similar problem can occur when returning a data structure containing a pointer.

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 612, “REALLOC (Reallocate Storage with New Length)” on page 785, and the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* 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

```
*.. 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.
 *
D DSbased         DS                    BASED(PTR1)
D    Field1                  1  16A
D     Field2                     2
D
D ARRAY           S            20A   DIM(12) BASED(PRT2)
D
D Temp_fld        S              *   BASED(PRT3)
D
D PTR2            S              *   INZ
D PTR3            S              *   INZ(*NULL)
```

Figure 97. 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             *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CL0N01++++++++++++++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 98. Pointer Arithmetic

Figure 99 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
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CL0N01++++++++++++++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 99. Obtaining a Julian Date

Figure 100 illustrates the use of pointers, based structures and system APIs. This program does the following:

1. Receives the Library and File name you wish to process
2. Creates a User space using the QUSCRTUS API
3. Calls an API (QUSLMBR) to list the members in the requested file
4. Gets a pointer to the User space using the QUSPTRUS API
5. Displays a message with the number of members and the name of the first and last member in the file

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D.....................................Keywords+++++++++++++++++++++++++++++
 *
D SPACENAME       DS
D                              10    INZ('LISTSPACE')
D                              10    INZ('QTEMP')
D ATTRIBUTE       S            10    INZ('LSTMBR')
D INIT_SIZE       S             9B 0 INZ(9999999)
D AUTHORITY       S            10    INZ('*CHANGE')
D TEXT            S            50    INZ('File member space')
D SPACE           DS                 BASED(PTR)
D SP1                       32767
 *
 * ARR is used with OFFSET to access the beginning of the
 * member information in SP1
 *
D ARR                           1    OVERLAY(SP1) DIM(32767)
 *
 * OFFSET is pointing to start of the member information in SP1
 *
D OFFSET                        9B 0 OVERLAY(SP1:125)
 *
 * Size has number of member names retrieved
 *
D SIZE                          9B 0 OVERLAY(SP1:133)
D MBRPTR          S              *
D MBRARR          S            10    BASED(MBRPTR) DIM(32767)
D PTR             S              *
D FILE_LIB        S            20
D  FILE           S            10
D  LIB            S            10
D WHICHMBR        S            10    INZ('*ALL      ')
D OVERRIDE        S             1    INZ('1')
D FIRST_LAST      S            50    INZ('    MEMBERS, +
D                                    FIRST =          , +
D                                    LAST =          ')
D IGNERR          DS
D                               9B 0 INZ(15)
D                               9B 0
D                               7A
```

Figure 100. Example of using pointers and based structures with an API (Part 1 of 2)

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CL0N01++++++++++++++Opcode(E)+Extended Factor 2++++++++++++++++++++++++++++
 *
 * Receive file and library you want to process
 *
C     *ENTRY        PLIST
C     FILE          PARM                    FILEPARM        10
C     LIB           PARM                    LIBPARM         10
 *
 * Delete the user space if it exists
 *
C                   CALL      'QUSDLTUS'                          10
C                   PARM                    SPACENAME
C                   PARM                    IGNERR
 *
 * Create the user space
 *
C                   CALL      'QUSCRTUS'
C                   PARM                    SPACENAME
C                   PARM                    ATTRIBUTE
C                   PARM                    INIT_SIZE
C                   PARM      ' '           INIT_VALUE       1
C                   PARM                    AUTHORITY
C                   PARM                    TEXT
 *
 * Call the API to list the members in the requested file
 *
C                   CALL      'QUSLMBR'
C                   PARM                    SPACENAME
C                   PARM      'MBRL0100'    MBR_LIST         8
C                   PARM                    FILE_LIB
C                   PARM                    WHICHMBR
C                   PARM                    OVERRIDE
 *
 * Get a pointer to the user-space
 *
C                   CALL      'QUSPTRUS'
C                   PARM                    SPACENAME
C                   PARM                    PTR
 *
 * Set the basing pointer for the member array
 * MBRARR now overlays ARR starting at the beginning of
 * the member information.
 *
C                   EVAL      MBRPTR = %ADDR(ARR(OFFSET))
C                   MOVE      SIZE          CHARSIZE         3
C                   EVAL      %SUBST(FIRST_LAST:1:3)  = CHARSIZE
C                   EVAL      %SUBST(FIRST_LAST:23:10) = MBRARR(1)
C                   EVAL      %SUBST(FIRST_LAST:41:10) = MBRARR(SIZE)
C     FIRST_LAST    DSPLY
C                   EVAL      *INLR = '1'
```

Figure 100. Example of using pointers and based structures with an API (Part 2 of 2)

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 101 on page 218 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)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CL0N01++++++++++++++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 101. How Not to Code Basing Pointers

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 16 bytes long and must be aligned on a 16 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 140.

The default initialization value for procedure pointers is *NULL.

Examples

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
  *
  * Define a basing pointer field and initialize to the address of the
  * data structure My_Struct.
  *
 D  My_Struct      DS
 D   My_array                    10    DIM(50)
 D
 D Ptr1            S             16*   INZ(%ADDR(My_Struct))
  *
  * Or equivalently, defaults to length 16 if length not defined
  *
 D Ptr1            S               *   INZ(%ADDR(My_Struct))
  *
  * Define a procedure pointer field and initialize to NULL
  *
 D Ptr1            S             16*   PROCPTR INZ(*NULL)
  *
  * Define a procedure pointer field and initialize to the address
  * of the procedure My_Proc.
  *
 D Ptr1            S             16*   PROCPTR INZ(%PADDR(My_Proc))
  *
  * Define pointers in a multiple occurrence data structure and map out
  * the storage.
  *
 DDataS          DS                    OCCURS(2)
 D ptr1                            *
 D ptr2                            *
 D Switch                         1A


 *
 * Storage map would be:
 *
 *
```

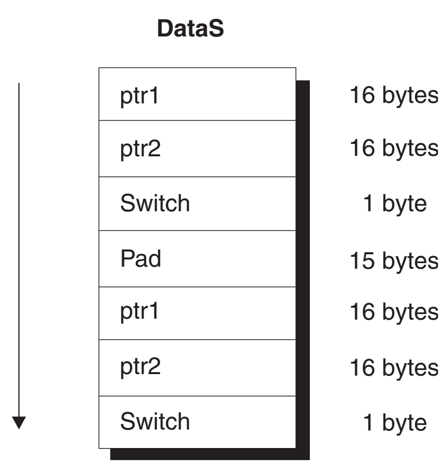


Figure 102. Defining pointers

Database Null Value Support

In an ILE 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 the ALWNULL keyword on a control specification is used (ALWNULL can also be specified as a command parameter):

1. ALWNULL(*USRCTL) - read, write, update, and delete records with null values and retrieve and position-to records with null keys.
2. ALWNULL(*INPUTONLY) - read records with null values to access the data in the null fields
3. 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 ALWNULL keyword.

User Controlled Support for Null-Capable Fields and Key Fields

When an externally described file contains null-capable fields and the ALWNULL(*USRCTL) keyword is specified on a control specification, 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 WORKSTN or PRINTER file, 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 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 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 an RPG 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, SEQ, WORKSTN 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.

Null-capable fields cannot be used as match fields or control-level fields.

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 103 on page 223 shows how to read, write and update records with null values when the ALWNULL(*USRCTL) option is used.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7....+....
 *
 *
 * Specify the ALWNULL(*USRCTL) keyword on a control
 * specification or compile the ILE RPG program with ALWNULL(*USRCTL)
 * on the command.
 *
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *H ALWNULL(*USRCTL)
 *
 * DISKFILE contains a record REC which has 2 fields: FLD1 and FLD2
 * Both FLD1 and FLD2 are null-capable.
 *
FDISKFILE  UF A E             DISK
 *
 * Read the first record.
 * Update the record with new values for any fields which are not
 * 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
 *
 * Read another record.
 * Update the record so that all fields are null.
 * There is no need to set the values of the fields because they
 * would be ignored.
C                   READ      REC                                    10
C                   EVAL      %NULLIND(Fld1) = *ON
C                   EVAL      %NULLIND(Fld2) = *ON
C                   UPDATE    REC
 *
 * Write a new record where Fld 1 is null and Fld 2 is not null.
 *
C                   EVAL      %NULLIND(Fld1) = *ON
C                   EVAL      %NULLIND(Fld2) = *OFF
C                   EVAL      Fld2 = 'New value'
C                   WRITE     REC
```

Figure 103. 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 104 and Figure 105 on page 225 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
  // 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 104. 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.
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
 C     Full_Kl       KLIST
 C                   KFLD                    Key1
 C                   KFLD      *IN02         Key2
 C                   KFLD      *IN03         Key3
 C     Partial_Kl    KLIST
 C                   KFLD                    Key1
 C                   KFLD      *IN05         Key2
  *
  * *IN02 is ON and *IN03 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 *IN02 is ON, the actual content in the search argument
  * for Key2 will be ignored.
  *
  * If a record exists in File1 with 'AA' in Key1, a null Key2, and
  * 'CC' in Key3, indicator 90 (the Eq indicator) will be set ON.
  *
 C                   MOVE      'AA'          Key1
 C                   MOVE      'CC'          Key3
 C                   EVAL      *IN02 = '1'
 C                   EVAL      *IN03 = '0'
 C     Full_Kl       SETLL     Rec1                                   90
```

Figure 105. Example of handling null key fields with KLIST (Part 1 of 2)

```
 *
 * The CHAIN operation below will retrieve a record with 'JJ' in Key1,
 * 'KK' in Key2, and a null Key3.  Again, because *IN03 is ON, even
 * if the programmer had moved some value (say 'XX') into the search
 * argument for Key3, 'XX' will not be used.  This means if File1
 * actually has a record with a key 'JJKKXX', that record will not
 * be retrieved.
 *
C                   MOVE      'JJ'          Key1
C                   MOVE      'KK'          Key2
C                   EVAL      *IN02 = '0'
C                   EVAL      *IN03 = '1'
C     Full_Kl       CHAIN     Rec1                                 80
 *
 * 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.
 *
 * In the database, the NULL value occupies the highest position in
 * the collating sequence.  Assume the keys in File1 are in ascending
 * sequence.  If File1 has a record with 'NN??xx' as key (where ??
 * means NULL and xx means any value other than NULL), that record
 * will be retrieved.  If such a record does not exist in File1, but
 * File1 has a record with 'NN????' as key, the 'NN????' record will
 * be retrieved.  The null flags for Key2 and Key3 will be set ON
 * as a result.
 *
C                   MOVE      'NN'          Key1
C                   SETON                                          05
C     Partial_Kl    CHAIN     Rec1                                 70
```

Figure 105. Example of handling null key fields with KLIST (Part 2 of 2)

Input-Only Support for Null-Capable Fields

When an externally described input-only file contains null-capable fields and the ALWNULL(*INPUTONLY) keyword is specified on a control specification, 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.
- Control-level indicators, match-field entries and 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.

Note: The same conditions apply for *INPUTONLY or *YES when specified on the ALWNULL command parameter.

ALWNULL(*NO)

When an externally described file contains null-capable fields and the ALWNULL(*NO) keyword is specified on a control specification, 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 ALWNULL(*USRCTL) option.

Error Handling for Database Data Mapping Errors

For any input or output operation, a data mapping error will cause a severe error message to be issued. For blocked output, if one or more of the records in the block contains data mapping errors and the file is closed before reaching the end of the block, a severe error message is issued and a system dump is created.

Chapter 10. Editing Numeric Fields

Editing provides a means of:

- Punctuating numeric fields, including the printing of currency symbols, commas, periods, minus sign, and floating minus
- Moving a field sign from the rightmost digit to the end of the field
- Blanking zero fields
- Managing spacing in arrays
- Editing numeric values containing dates
- Floating a currency symbol
- Filling a print field with asterisks

This chapter applies only to non-float numeric fields. To output float fields in the external display representation, specify blank in position 52 of the output specification. To obtain the external display representation of a float value in calculations, use the %EDITFLT built-in function.

A field can be edited by edit codes, or edit words. You can print fields in edited format using output specifications or you can obtain the edited value of the field in calulation specifications using the built-in functions %EDITC (edit code) and %EDITW (edit word).

When you print fields that are not edited, the fields are printed as follows:

- Float fields are printed in the external display representation.
- Other numeric fields are printed in zoned numeric representation.

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)	004765K	004765K	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 K, which is not numeric. The K 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. Therefore, to improve the readability of the printed output, numeric fields should 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 three categories: simple (X, Y, Z), combination (1 through 4, A through D, J through Q), and user-defined (5 through 9). In output specifications, you enter the edit code in position 44 of the field to be edited. In calculation specifications, you specify the edit code as the second parameter of the %EDITC built-in function.

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 F sign for positive fields and a hexadecimal D sign for negative fields. However, because the system does this, you normally do not have to specify this code. Leading zeros are not suppressed. You can use %EDITC with the X edit code to convert a number to character with leading zeros. However, be aware that negative numbers can produce unexpected results; for example, %EDITC(-00123:'X') will give the result '0012L'.
- 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})” on page 262 and “DECEDIT(*JOBRUN | 'value')” on page 264 keywords on the control specification can be used to alter edit formats.

  Note: 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.

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 zeros are suppressed. The decimal position of the source field determines whether and where a decimal point is placed. If decimal positions are specified for the source field and the zero balance is to be suppressed, the decimal separator is included *only* if the field is not zero. If a zero balance is to be suppressed, a zero field is output as blanks.

When a zero balance is not to be suppressed and the field is equal to zero, either of the following is output:

- 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. 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.

The currency symbol for the program is a dollar sign ($) unless a currency symbol is specified with the CURSYM keyword on the control specification.

To specify a floating currency symbol in output specifications, code the currency symbol in positions 53-55 as well as an edit code in position 44 for the field to be edited.

For 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.

Asterisk protection causes an asterisk to replace each zero suppressed. A complete field of asterisks replaces the fiield on a zero balance source field. To specify asterisk protection in output specifications, code an asterisk constant in positions 53 through 55 of the output specifications, along with an edit code. 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) or with the user-defined (5 through 9) edit codes.

A currency symbol can appear before the asterisk fill (fixed currency symbol). You can do this in 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.

```
C          EVAL      X = '$' + %EDITC(N: 'A' : *ASTFILL)
```

In output specifications, 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 38 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 38. 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

User-Defined Edit Codes

IBM has predefined edit codes 5 through 9. You can use them as they are, or you can delete them and create your own. For a description of the IBM-supplied edit codes, see the iSeries Information Center programming category.

The user-defined edit codes allow you to handle common editing problems that would otherwise require the use of an edit word. Instead of the repetitive coding of the same edit word, a user-defined edit code can be used. These codes are system defined by the CL command CRTEDTD (Create Edit Description).

When you edit a field defined to have decimal places, be sure to use an edit word that has an editing mask for both the fractional and integer portions of the field. Remember that when a user-defined edit code is specified in a program, any system changes made to that user-defined edit code are not reflected until the program is recompiled. For further information on CRTEDTD, see the iSeries Information Center programming category.

Editing Considerations

Remember the following when you specify any of the edit codes:

- 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 edited value. 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 39 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 40 on page 234 shows how fields look after they are edited.

Table 41 on page 235 shows the effect that the different edit codes have on the same field with a specified end position for output.

Table 39. 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

Table 39. Edit Codes (continued)

				DECEDIT Keyword Parameter				
Edit Code	**Commas**	**Decimal Point**	**Sign for Negative Balance**	**'.'**	**','**	**'0,'**	**'0.'**	**Zero Suppress**
5-9[1]								
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	- (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[2]								
Y[3]								Yes
Z[4]								Yes

Notes:

1. These are the user-defined edit codes.
2. The X edit code ensures a hexadecimal F sign for positive values. Because the system does this for you, normally you do not have to specify this code.
3. 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
```

4. The Z edit code removes the sign (plus or minus) from a numeric field and suppresses leading zeros.

Table 40. 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	00012ƀ[5]	00012ƀ[5]	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		
5-9[1]						
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[2]	1234567	1234567	00012ƀ[5]	00012ƀ[5]	000000	000000
Y[3]			0/01/20	0/01/20	0/00/00	0/00/00
Z[4]	1234567	1234567	120	120		

Notes:

1. These edit codes are user-defined.
2. 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.
3. 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
```

4. The Z edit code removes the sign (plus or minus) from a numeric field and suppresses leading zeros of a numeric field.
5. The ƀ represents a blank. This may occur if a negative zero does not correspond to a printable character.

Table 41. 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**	**11**
Unedited				0	0	4	1	K[1]	
1					4	.	1	2	
2					4	.	1	2	
3					4	.	1	2	
4					4	.	1	2	
5-9[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	-	
K				4	.	1	2	-	
L				4	.	1	2	-	
M				4	.	1	2	-	
N				-	4	.	1	2	
O				-	4	.	1	2	
P				-	4	.	1	2	
Q				-	4	.	1	2	
X				0	0	4	1	K[1]	
Y			0	/	4	1	/	2	
Z						4	1	2	

Notes:

1. K represents a negative 2.
2. These are user-defined edit codes.

Edit Words

If you have editing requirements that cannot be met by using the edit codes described above, 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 an 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, which the system applies to the source data to produce the output.

The edit word may be specified directly on an output specification or may 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 calulation specifications using the built-in function %EDITW (edit word).

Edit words are limited to 115 characters.

How to Code an Edit Word

To output using 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 consists of three parts: the body, the status, and the expansion. The following shows the three parts of an edit word:

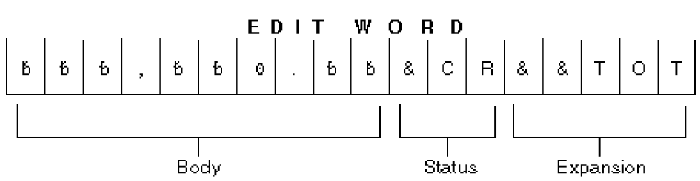


Figure 106. Parts of an Edit Word

The *body* is the space for the digits transferred from the source data field to the edited result. 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:

Blank: Blank is replaced with the character from the corresponding position of the value to be edited. A blank position is referred to as a digit position.

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 appear unless there is a significant digit to its left.

Edit Word	Source Data	Appears in Edited Result as:
'ƀƀƀƀƀƀƀ'	0000072	ƀƀƀƀƀ72
'ƀƀƀƀƀƀƀ.ƀƀ'	000000012	ƀƀƀƀƀƀƀƀ12
'ƀƀƀƀƀƀƀ.ƀƀ'	000000123	ƀƀƀƀƀƀ1.23

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” below).

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 Edited Result as:
'ƀƀƀ0ƀƀƀƀƀƀ'	00000004	ƀƀƀƀ000004
'ƀƀƀ0ƀƀƀƀƀƀ'	012345	ƀƀƀƀ012345
'ƀƀƀ0ƀƀƀƀƀƀ'	012345678	ƀƀ12345678

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 Edited Result as:
'0ƀƀƀ'	0156	ƀ156
'0ƀƀƀƀ'	0156	ƀ0156

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 Edited Result as:
'ƀƀƀƀƀƀ0.ƀƀ'	000000001	ƀƀƀƀƀƀƀ.01
'ƀƀƀƀƀƀ0.ƀƀ'	000000000	ƀƀƀƀƀƀƀ.00
'ƀƀƀ,ƀ0ƀ.ƀƀ'	00000012	ƀƀƀƀƀƀ0.12
'ƀƀƀ,ƀ0ƀ.ƀƀ'	00000123	ƀƀƀƀƀƀ1.23
'ƀ0ƀ,ƀƀƀ.ƀƀ'	00000123	ƀƀ0,001.23

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 above for “Zeros”.

Edit Word	Source Data	Appears in Edited Result as:
'*ƀƀƀƀƀƀ.ƀƀ'	000000123	*ƀƀƀƀƀ1.23
'ƀƀƀƀƀ*ƀ.ƀƀ'	000000000	******0.00
'ƀƀƀƀƀ*ƀ.ƀƀ**'	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.

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 Edited Result as:
'ƀƀ,ƀƀƀ,ƀ$0.ƀƀ'	000000012	ƀƀƀƀƀƀƀƀƀ$.12
'ƀƀ,ƀƀƀ,ƀ$0.ƀƀ'	000123456	ƀƀƀƀ$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 Edited Result as:
'$ƀ,ƀƀƀ,ƀƀ0.ƀƀ'	000123456	$ƀƀƀƀ1,234.56
'$ƀƀ,ƀƀƀ,0ƀ0.ƀƀ'	000000000	$ƀƀƀƀƀƀƀƀ00.00
'$ƀ,ƀƀƀ,*ƀƀ.ƀƀ'	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" below).

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 Edited Result as:
'0AREA&ƀƀƀ&NO.&ƀƀƀ-ƀƀƀƀ'	4165551212	ƀAREAƀ416ƀNO.ƀ555-1212

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 Edited Result as:
'$ƀƀƀƀƀƀ**DOLLARS&ƀƀ&CTS'	000012345	$****123*DOLLARSƀ45ƀCTS
'$ƀƀƀƀƀƀ**DOLLARS&ƀƀ&CTS'	000000006	$********DOLLARSƀ06ƀCTS
'$ƀƀƀƀƀƀƀ&DOLLARS&ƀƀ&CTS'	000000006	$ƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀ6ƀCTS

A date could be edited by using either edit word:

Edit Word	Source Data	Appears in Edited Result as:
'ƀƀ/ƀƀ/ƀƀ'	010388	ƀ1/03/88
'0ƀƀ/ƀƀ/ƀƀ'	010389	ƀ01/03/89

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 Edited Result as:
'ƀƀ0.ƀƀ000'	01234	ƀ12.34000
'ƀƀ*.ƀƀ000'	01234	*12.34000

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 Edited Result as:
'ƀƀƀƀƀƀƀ.ƀƀ-'	000000123-	ƀƀƀƀƀƀ1.23-
'ƀƀƀƀƀƀƀ.ƀƀ-'	000000123	ƀƀƀƀƀƀ1.23ƀ

Constants between the last replaceable character and the - or CR symbol will print only if the field is negative; otherwise, blanks will appear in these positions. Note the use of ampersands to represent blanks:

Edit Word	Source Data	Appears in Edited Result as:
'ƀ,ƀƀƀ,ƀƀ0.ƀƀ&30&DAY&CR'	000000123-	ƀƀƀƀƀƀƀƀƀ1.23ƀ30ƀDAYƀCR
'ƀ,ƀƀƀ,ƀƀ0.ƀƀ&30&DAY&CR'	000000123	ƀƀƀƀƀƀƀƀƀ1.23ƀƀƀƀƀƀƀƀƀƀ

Formatting the Expansion of an Edit Word

The characters in the expansion portion of an edit word are always used. The expansion cannot contain blanks. If a blank is required in the edited result, specify an ampersand in the body of the edit word.

Constants may be added to appear with any value of the number:

Edit Word	Source Data	Appears in Edited Result as:
'ƀ,ƀƀ0.ƀƀ&CR&NET'	000123-	ƀƀƀƀ1.23ƀCRƀNET
'ƀ,ƀƀ0.ƀƀ&CR&NET'	000123	ƀƀƀƀ1.23ƀƀƀƀNET

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 Edited Result as:
'ƀƀ0.ƀƀ&SECRET'	12345-	123.45ƀSECRET
'ƀƀ0.ƀƀ&SECRET'	12345	123.45ƀƀƀƀƀET
'ƀƀ0.ƀƀ&CR&&SECRET'	12345	123.45ƀƀƀƀƀSECRET

Summary of Coding Rules for Edit Words

The following rules apply to edit words in output specifications:

- 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 following rules apply to edit words in general:

- 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 externally described files, place the edit codes in data description specifications (DDS), instead of in RPG IV specifications. See the iSeries Information Center database and file systems category for information on how to specify edit codes in the data description specifications. However, if an externally described file, which has an edit code specified, is to be written out as a program described output file, you must specify editing in the output specifications. In this case, any edit codes in the data description specifications are ignored.

Part 3. Specifications

This section describes the RPG IV specifications. First, information common to several specifications, such as keyword syntax and continuation rules is described. Next, the specifications are described in the order in which they must be entered in your program. Each specification description lists all the fields on the specification and explains all the possible entries.

Chapter 11. About Specifications

RPG IV source is coded on a variety of specifications. Each specification has a specific set of functions.

This reference contains a detailed description of the individual RPG IV specifications. Each field and its possible entries are described. Chapter 19, “Operations,” on page 423 describes the operation codes that are coded on the calculation specification, which is described in Chapter 16, “Calculation Specifications,” on page 391.

RPG IV Specification Types

There are three groups of source records that may be coded in an RPG IV program: the main source section, the subprocedure section, and the program data section. The **main source section** consists of the first set of H, F, D, I, C, and O specifications in a module. If MAIN or NOMAIN is specified on a Control specification, this section does not contain a cycle-main procedure, and so it cannot contain any executable calculations. If the keyword MAIN or NOMAIN is not specified, this corresponds to a standalone program or a cycle-main procedure. Every module requires a main source section independently of whether subprocedures are coded.

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 RPG IV language consists of a mixture of position-dependent code and free form code. Those 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 an extended-factor 2. Otherwise, RPG IV entries are position specific. To represent this, each illustration of RPG IV code will be in listing format with a scale drawn across the top.

The following illustration shows the types of source records that may be entered into each group and their order.

Note

The RPG IV source must be entered into the system in the order shown in Table 42. Any of the specification types can be absent, but at least one from the main source section must be present.

Table 42. Source Records and Their Order in an RPG IV Source Program

Main Source Section H Control F File Description D Definition I Input C Calculation O Output
Subprocedure Section *(Repeated for each procedure)* P Procedure F File Description D Definition C Calculation P Procedure
Program Data when the ** form is used ** File Translation Records ** Alternate Collating Sequence Records ** Compile-Time Array and Table Data
Program Data when the **TYPE form is used *(Specified in any order)* **CTDATA ARRAY1 Compile-Time Array Data **FTRANS File Translation Records **CTDATA TABLE2 Compile-Time Table Data **ALTSEQ Alternate Collating Sequence Records **CTDATA ARRAY3 Compile-Time Array Data

Main Source Section Specifications

H Control (Header) specifications provide information about program generation and running of the compiled program. Refer to Chapter 12, “Control Specifications,” on page 255 for a description of the entries on this specification.

F File description specifications define the global files for the program. Refer to Chapter 13, “File Description Specifications,” on page 279 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, prototypes and their parameters, and procedure interfaces and their parameters are defined on this specification. Refer to Chapter 14, “Definition Specifications,” on page 315 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 15, “Input Specifications,” on page 375 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. Refer to Chapter 16, “Calculation Specifications,” on page 391 for a description of the entries on this specification.

O Output specifications describe the records and fields and indicate when they are to be written by the program. Refer to Chapter 17, “Output Specifications,” on page 401 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 18, “Procedure Specifications,” on page 417 for a description of the entries on this specification.

F File description specifications define the files used locally in the subprocedure. Refer to Chapter 13, “File Description Specifications,” on page 279for 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 14, “Definition Specifications,” on page 315 for a description of the entries on this specification.

C Calculation specifications perform the logic of the prototyped procedure. Refer to Chapter 16, “Calculation Specifications,” on page 391 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 any of these keywords as required: “CTDATA” on page 326, “FTRANS{(*NONE | *SRC)}” on page 268, or “ALTSEQ{(*NONE | *SRC | *EXT)}” on page 258. By associating the program data with the appropriate 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 associated 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 164. For more information on file translation, see “File Translation” on page 118. For more information on alternate collating sequences, see “Alternate Collating Sequence” on page 195.

Common Entries

The following entries are common to all RPG specifications preceding program data:

- Positions 1-5 can be used for comments.
- Specification type (position 6). The following letter codes can be used:

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 will denote 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 to 80 are blank.
- Positions 7 to 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 43. 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 parameters.	FLTDIV
blank	OPTIONS(*OMIT *NOPASS *VARSIZE *STRING *TRIM *RIGHTADJ)	One of *OMIT, *NOPASS, *VARSIZE, *STRING, *TRIM, or *RIGHTADJ is required and more than one parameter can be optionally specified.	OPTIONS(*OMIT : *NOPASS : *VARSIZE : *TRIM : *RIGHTADJ)

Continuation Rules

The fields that 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 non-blank 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.
    - Each segment of the literal must be enclosed by shift-out and shift-in characters.
    - When the a graphic literal is assembled, only the first shift-out and the last shift-in character will be included.
    - Regardless of which continuation character is used for a graphic literal, the literal continues with the first character after the shift-out character on the continuation line. Spaces preceding the shift-out character are ignored.
  – 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 non-blank 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 253.
  – 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.

  Example

```
*.. 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
CL0N01Factor1+++++++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)
```

Control Specification Keyword Field

The rule for continuation on the control specification is:

- The specification continues on or past position 7 of the next control specification

  Example

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DATFMT(
H      *MDY&
H      )
```

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

  Example

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++++
F.....................................Keywords++++++++++++++++++++++++++++
F                                     EXTIND
F                                     (
F                                     *INU1
F                                     )
```

Definition Specification Keyword Field

The rules for continuation of keywords on the definition specification are:

- The specification continues on or past position 44 of the next Definition specification dependent on the continuation character specified
- Positions 7-43 of the continuation line must be blank

Example

```
*.. 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 -
 * Only a comment or a completely blank line is allowed in here
D                                     fleece was white as snow.'
D                                     )
 * Numeric literal, continues with the first non blank in/past position 44
 *
DNUMERIC         C                    12345
D                                       67
 * Graphic named constant, must have shift-out in/past position 44
DGRAF            C                    G'oAABBCCDDi+
D                                      oEEFFGGi'
```

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

Example

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++++
C                             Extended-factor2-++++++++++++++++++++++++++++
C                   EVAL      MARY='Mary had a little lamb, its +
 * Only a comment or a completely blank line is allowed in here
C                                fleece was white as snow.'
 *
 *  Arithmetic expressions do not have continuation characters.
 *  The '+' sign below is the addition operator, not a continuation
 *  character.
C
C                   EVAL      A = (B*D)/ C +
C                             24
 *  The first use of '+' in this example is the concatenation
 *  operator.  The second use is the character literal continuation.
C                   EVAL      ERRMSG = NAME +
C                                      ' was not found +
C                                      in the file.'
```

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

  Example

```
*.. 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-

 *
 * Only a comment or a completely blank line is allowed in here
O                                             fleece was white as snow.'
```

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.

     Example

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
DName+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D                                      Keywords-cont++++++++++++++++++++++++
* Long name without continued name lines:
D RatherLongName  S            10A
* Long name using 1 continued name line:
D NameThatIsEvenLonger...
D                 C                  'This is the constant -
D                                    that the name represents.'
* Long name using 1 continued name line:
D NameThatIsSoLongItMustBe...
D   Continued     S            10A
* Compile-time arrays may have long names:
D CompileTimeArrayContainingDataRepresentingTheNamesOfTheMonthsOf...
D  TheYearInGermanLanguage...
D                 S            20A   DIM(12) CTDATA PERRCD(1)
* Long name using 3 continued name lines:
D ThisNameIsSoMuchLongerThanThe...
D    PreviousNamesThatItMustBe...
D    ContinuedOnSeveralSpecs...
D                 PR           10A
D parm_1                       10A   VALUE
*
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++++
C                             Extended-factor2-++++++++++++++++++++++++++++
* Long names defined on calc spec:
C     LongTagName   TAG
C     *LIKE         DEFINE    RatherLongNameQuiteLongName    +5
*
PName+++++++++++..B...................Keywords+++++++++++++++++++++++++++++
PContinuedName+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
* 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
```

Chapter 12. Control Specifications

The control-specification statements, identified by an H in position 6, provide information about generating and running programs. However, there are three different ways in which this information can be provided to the compiler and the compiler searches for this information in the following order:

1. A control specification included in your source
2. A data area named RPGLEHSPEC in *LIBL
3. A data area named DFTLEHSPEC in QRPGLE

Once one of these sources is found, the values are assigned and keywords that are not specified are assigned their default values.

See the description of the individual entries for their default values.

Note: Compile-option keywords do not have default values. The keyword value is initialized with the value you specify for the CRTBNDRPG or CRTRPGMOD command.

TIP

The control specification keywords apply at the module 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.

Using a Data Area as a Control Specification

Use the CL command CRTDTAARA (Create Data Area) to create a data area defined as type *CHAR. (See the iSeries Information Center programming category for a description of the Create Data Area command.) Enter the keywords and their possible parameters that are to be used in the Initial Value field of the command.

For example, to create an RPGLEHSPEC data area that will specify a default date format of *YMD, and a default date separator /, you would enter:

```
CRTDTAARA DTAARA(MYLIB/RPGLEHSPEC)
          TYPE(*CHAR)
          LEN(80)
          VALUE('datfmt(*ymd) datedit(*ymd/)')
```

The data area can be whatever size is required to accommodate the keywords specified. The entire length of the data area can only contain keywords.

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 107. Control-Specification Layout

The following is an example of a control specification.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H ALTSEQ(*EXT) CURSYM('$') DATEDIT(*MDY) DATFMT(*MDY/) DEBUG(*YES)
H DECEDIT('.') FORMSALIGN(*YES) FTRANS(*SRC) DFTNAME(name)
H TIMFMT(*ISO)
H COPYRIGHT('(C) Copyright ABC Programming - 1995')
```

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 will deal with devices and how certain types of information will be represented.

The control-specification keywords also include compile-option keywords that override the default or specified options on the CRTBNDRPG and CRTRPGMOD commands. These keywords determine the compile options to be used on every compile of the program.

Control-Specification 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 255 and “Control Specification Keyword Field” on page 251.

ALLOC(*STGMDL | *TERASPACE | *SNGLVL)

The ALLOC keyword specifies the storage model for storage management operations in the module.

If the ALLOC keyword is not specified, ALLOC(*STGMDL) is assumed.

- *STGMDL is used to specify that the storage model for memory management operations will be the same as the storage model of the module. You use the STGMDL keyword on the Control specification to control the storage model of the module. If the storage model of the module is *INHERIT, the storage model used for memory management operations is determined at runtime.
- *SNGLVL is used to specify that the single-level storage model will be used for memory management operations.
- *TERASPACE is used to specify that the teraspace storage model will be used for memory management operations.

See “Memory Management Operations” on page 458 for more information on teraspace and single-level memory management operations.

ACTGRP(*STGMDL | *NEW | *CALLER | 'activation-group-name')

The ACTGRP keyword allows you to specify the activation group the program is associated with when it is called. If ACTGRP(*STGMDL) is specified and STGMDL(*SNGLVL) or STGMDL(*INHERIT) is in effect, the program will be activated into the QILE activation group when it is called. If ACTGRP(*STGMDL) is specified and STGMDL(*TERASPACE) is in effect, the program will be activated into the QILETS activation group when it is called. If ACTGRP(*NEW) is specified, then the program is activated into a new activation group. If ACTGRP(*CALLER) is specified, then the program is activated into the caller's activation group. If an activation-group-name is specified, then that name is used when this program is called.

If the ACTGRP keyword is not specified, then the value specified on the command is used.

The ACTGRP keyword is valid only if the CRTBNDRPG command is used.

You cannot use the ACTGRP, BNDDIR, or STGMDL keywords when creating a program with DFTACTGRP(*YES).

Note: The name of the activation group created when the program is called will have exactly the same case as the text entered for the activation-group-name. The RCLACTGRP command does not allow lower-case text to be specified for its ACTGRP parameter. If it is required to reclaim an activation group individually using the RCLACTGRP command then do not enter lower-case case text for the activation-group-name.

ALTSEQ{(*NONE | *SRC | *EXT)}

The ALTSEQ keyword indicates whether an alternate collating sequence is used, if so, whether it is internal or external to the source. The following list shows what happens for the different possible keyword and parameter combinations.

Keyword/Parameter
: **Collating Sequence Used**

ALTSEQ not specified
: Normal collating sequence

ALTSEQ(*NONE)
: Normal collating sequence

ALTSEQ, no parameters
: Alternate collating sequence specified in source

ALTSEQ(*SRC)
: Alternate collating sequence specified in source

ALTSEQ(*EXT)
: Alternate collating sequence specified by the SRTSEQ and LANGID command parameters or keywords.

If ALTSEQ is not specified or specified with *NONE or *EXT, an alternate collating sequence table must not be specified in the program.

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 219

AUT(*LIBRCRTAUT | *ALL | *CHANGE | *USE | *EXCLUDE | 'authorization-list-name')

The AUT keyword specifies the authority given to users who do not have specific authority to the object, who are not on the authorization list, and whose user group has no specific authority to the object. The authority can be altered for all users or for specified users after the object is created with the CL commands Grant Object Authority (GRTOBJAUT) or Revoke Object Authority (RVKOBJAUT).

If AUT(*LIBRCRTAUT) is specified, then the public authority for the object is taken from the CRTAUT keyword of the target library (the library that contains the object). The value is determined when the object is created. If the CRTAUT value for the library changes after the create, the new value will not affect any existing objects.

If AUT(*ALL) is specified, then authority is provided for all operations on the object, except those limited to the owner or controlled by authorization list management authority. The user can control the object's existence, specify this security for it, change it, and perform basic functions on it, but cannot transfer its ownership.

If AUT(*CHANGE) is specified, then it provides all data authority and the authority to perform all operations on the object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it.

If AUT(*USE) is specified, then it provides object operational authority and read authority; that is, authority for basic operations on the object. The user is prevented from changing the object.

If AUT(*EXCLUDE) is specified, then it prevents the user from accessing the object.

The authorization-list-name is the name of an authorization list of users and authorities to which the object is added. The object will be secured by this authorization list, and the public authority for the object will be set to *AUTL. The authorization list must exist on the system at compilation time.

If the AUT keyword is not specified, then the value specified on the command is used.

BNDDIR('binding-directory-name' {:'binding-directory-name'...})

The BNDDIR keyword specifies the list of binding directories that are used in symbol resolution.

A binding directory name can be qualified by a library name followed by a slash delimiter ('library-name/binding-directory-name'). The library name is the name of the library to be searched. If the library name is not specified, *LIBL is used to find the binding directory name. When creating a program using CRTBNDRPG, the library list is searched at the time of the compile. When creating a module using CRTRPGMOD, the library list is searched when the module is used to create a program or service program.

If BNDDIR is specified on both the control specification and on the command, all binding directories are used for symbol resolution. The BNDDIR on the control specification does not override the BNDDIR on the command.

If the BNDDIR keyword is not specified, then the value specified on the command is used.

You cannot use the BNDDIR, ACTGRP, or STGMDL command parameters or keywords when creating a program with DFTACTGRP(*YES).

CCSID(*GRAPH : parameter | *UCS2 : number | *CHAR : *JOBRUN)

CCSID(*GRAPH) and CCSID(*UCS2) set the default graphic (*GRAPH) and UCS-2 (*UCS2) CCSIDs for the module. These defaults are used for literals, compile-time data, program-described input and output fields, and data definitions that do not have the CCSID keyword coded.

CCSID(*CHAR) sets the CCSID used for the module's character data at runtime.

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. A valid graphic CCSID is 65535 or a CCSID with the EBCDIC double-byte encoding scheme (X'1200').

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. A valid UCS-2 CCSID has the UCS-2 encoding scheme (x'7200'). For example, the UTF-16 CCSID 1200 has encoding scheme x'7200'.

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 325.)
- Program-described graphic or UCS-2 input and output fields and keys are assumed to have the module's default CCSID.

CCSID(*CHAR : *JOBRUN)
When CCSID(*CHAR:*JOBRUN) is specified, character data will be assumed to be in the job CCSID at runtime. The character X'0E' will be assumed to be a shift-out character only if the runtime job CCSID is a mixed-byte CCSID.

When CCSID(*CHAR : *JOBRUN) is not specified, character data will be assumed to be in the mixed-byte CCSID related to the job CCSID. If the character X'0E' appears in character data, it will be interpreted as a shift-out character. This may cause incorrect results when character data is converted to UCS-2 data.

Note: Specifying CCSID(*CHAR:*JOBRUN) does not change the behaviour of the compiler with respect to character literals containing X'0E'. When a character literal contains X'0E', the compiler will always treat it as a shift-out character, independent of the CCSID(*CHAR) keyword.

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 that can be seen using the DSPMOD, DSPPGM, or DSPSRVPGM commands. 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 249 for rules on using continuation lines.) If the COPYRIGHT keyword is not specified, copyright information is not added to the created module or program.

TIP

To see the copyright information for a module, use the command:

```
DSPMOD mylib/mymod DETAIL(*COPYRIGHT)
```

For a program, use the DSPPGM command with DETAIL(*COPYRIGHT). This information includes the copyright information from all modules bound into the program.

Similarly, DSPSRVPGM DETAIL(*COPYRIGHT) gives the copyright information for all modules in a service 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 RPG character set (see Chapter 1, "Symbolic Names and Reserved Words," on page 3) may be used except:

- 0 (zero)
- * (asterisk)
- , (comma)
- & (ampersand)
- . (period)
- – (minus sign)
- C (letter C)
- R (letter R)
- Blank

If the keyword is not specified, $ (dollar sign) will be used as the currency symbol.

CVTOPT(*{NO}DATETIME *{NO}GRAPHIC *{NO}VARCHAR *{NO}VARGRAPHIC)

The CVTOPT keyword is used to determine how the ILE RPG compiler handles date, time, timestamp, graphic data types, and 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 also be used, and vice versa. For example, if you specify *GRAPHIC you cannot also specify *NOGRAPHIC, 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(*DATETIME : *NOVARCHAR : *NOVARGRAPHIC) is specified on the Control specification, then for the pair (*GRAPHIC, *NOGRAPHIC), whatever was specified implicitly or explicitly on the command will be used.

If *DATETIME is specified, then date, time, and timestamp data types are declared as fixed-length character fields.

If *NODATETIME is specified, then date, time, and timestamp data types are not converted.

If *GRAPHIC is specified, then double-byte character set (DBCS) graphic data types are declared as fixed-length character fields.

If *NOGRAPHIC is specified, then double-byte character set (DBCS) graphic types are not converted.

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 internal 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 keyword on the definition specification for that field.

If the DATFMT keyword is not specified, the *ISO format is assumed. For more information on internal formats, see “Internal and External Formats” on page 179. Table 33 on page 207 describes the various date formats and their separators.

DEBUG{(*INPUT | *DUMP | *XMLSAX | *NO | *YES)}

The DEBUG keyword controls what debugging aids are generated into the module.

When the DEBUG keyword is specified with one or more of the *INPUT, DUMP or *XMLSAX parameters, you can choose exactly which debugging aids are to be generated into the module. When the DEBUG keyword is specified with *YES or *NO, no other parameters can be specified.

***INPUT**
All externally described input fields will be read during input operations even if they are not used in the program. Normally, externally described input fields are only read during input operations if the field is otherwise used within the program.

***DUMP**
DUMP operations are performed.

Note: You can force a DUMP operation to be performned by specifying operation extender A on the DEBUG operation code. This operation extender means that a dump is always performed, regardless of the value of the DEBUG keyword.

***XMLSAX**
An array with the name _QRNU_XMLSAX will be generated into the module if it has a debug view (if it is compiled with a value for the DBGVIEW parameter other than *NONE). The values of the array will be the names of the *XML special words, without the "*XML_" prefix. For example, if *XML_START_DOCUMENT has the value 1, _QRNU_XMLSAX(1) will have the value "START_DOCUMENT".

Sample debug session:

```
> EVAL event
  EVENT = 2
> EVAL _QRNU_XMLSAX(event)
  _QRNU_XMLSAX(EVENT) = 'END_DOCUMENT   '
```

Specifying the DEBUG keyword with *NO indicates that no debugging aids should be generated into the module. This is the same as omitting the DEBUG keyword entirely. No other parameters can be specified when *NO is specified.

Specifying the DEBUG keyword with *YES or with no parameters is the same as specifying DEBUG(*INPUT : *DUMP). No other parameters can be specified when *YES is specified. The value *YES is retained for compatibility; it is preferable to specify the more granular values *INPUT, *DUMP and *XMLSAX.

Examples:

```
  * 1. All of the debugging aids are available
 H DEBUG(*INPUT : *DUMP : *XMLSAX)
* 2. None of the debugging aids are available
 H DEBUG(*NO)
* 3. Only the debugging aid related to input fields is available
 H DEBUG(*INPUT)
* 4. The debugging aids related to the DUMP operation and
*    to XML-SAX parsing are available
 H DEBUG(*XMLSAX : *DUMP)
```

Note: The DEBUG keyword does not control whether the module is created to be debuggable. That is controlled by the DBGVIEW parameter for the CRTBNDRPG or CRTRPGMOD command. The DEBUG keyword controls additional debugging aids.

DECEDIT(*JOBRUN | 'value')

The DECEDIT keyword specifies the character used as the decimal point for edited decimal numbers and whether or not leading zeros are printed.

If *JOBRUN is specified, the DECFMT value associated with the job at runtime is used. The possible job decimal formats are listed in the following table:

*Table 44. DECEDIT with *JOBRUN*

Job Decimal Format	Decimal Point	Print Leading Zeros	Edited Decimal Number
blank	period (.)	No	.123
I	comma (,)	No	,123
J	comma (,)	Yes	0,123

If a value is specified, then the edited decimal numbers are printed according to the following possible values:

Table 45. DECEDIT with 'value'

'Value'	Decimal Point	Print Leading Zeros	Edited Decimal Number
'.'	period (.)	No	.123
','	comma (,)	No	,123
'0.'	period (.)	Yes	0.123
'0,'	comma (,)	Yes	0,123

If DECEDIT is not specified, a period (.) is used for editing numeric values.

Note: Zeros to the right of a decimal point are always printed.

DECPREC(30|31|63)

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 expressions are presented when used in %EDITC, %EDITW, %CHAR, %LEN, and %DECPOS.

DECPREC(30)
The default decimal precision. It indicates that the maximum precision of

decimal values when used in the affected operations is 30 digits. However, if at least one operand in the expression is a decimal variable with 31 digits, DECPREC(31) is assumed for that expression. If at least one operand in the expression is a decimal variable with 32 or more digits, DECPREC(63) is assumed for that expression.

DECPREC(31)
The maximum precision of decimal values when used in the affected operations is 31 digits. However, if at least one operand in the expression is a decimal variable with 32 digits or more, DECPREC(63) is assumed for that expression.

DECPREC(63)
The number of digits used in the affected operations is always computed following the normal rules for decimal precision, which can be up to the maximum of 63 digits.

DFTACTGRP(*YES | *NO)

The DFTACTGRP keyword specifies the activation group in which the created program will run when it is called.

If *YES is specified, then this program will always run in the default activation group, which is the activation group where all original program model (OPM) programs are run. This allows ILE RPG programs to behave like OPM RPG programs in the areas of file sharing, file scoping, RCLRSC, and handling of unmonitored exceptions. ILE static binding is not available when a program is created with DFTACTGRP(*YES). This means that you cannot use the BNDDIR, ACTGRP, or STGMDL command parameters or keywords when creating this program. In addition, any call operation in your source must call a program and not a procedure. DFTACTGRP(*YES) is useful when attempting to move an application on a program-by-program basis to ILE RPG.

If *NO is specified, then the program is associated with the activation group specified by the ACTGRP command parameter or keyword and static binding is allowed. DFTACTGRP(*NO) is useful when you intend to take advantage of ILE concepts; for example, running in a named activation group or binding to a service program.

If the DFTACTGRP keyword is not specified, then the value specified on the command is used.

The DFTACTGRP keyword is valid only if the CRTBNDRPG command is used.

DFTNAME(rpg_name)

The DFTNAME keyword specifies a default program or module name. When *CTLSPEC is specified on the create command, the rpg_name is used as the program or module name. If rpg_name is not specified, then the default name is RPGPGM or RPGMOD for a program or module respectively. The RPG rules for names (see “Symbolic Names” on page 3) apply.

ENBPFRCOL(*PEP | *ENTRYEXIT | *FULL)

The ENBPFRCOL keyword specifies whether performance collection is enabled.

If *PEP is specified, then performance statistics are gathered on the entry and exit of the program-entry procedure only. This applies to the actual program-entry procedure for an object, not to the main procedure of the object within the object.

If *ENTRYEXIT is specified, then performance statistics are gathered on the entry and exit of all procedures of the object.

If *FULL is specified, then performance statistics are gathered on entry and exit of all procedures. Also, statistics are gathered before and after each call to an external procedure.

If the ENBPFRCOL keyword is not specified, then the value specified on the command is used.

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.

FIXNBR(*{NO}ZONED *{NO}INPUTPACKED)

The FIXNBR keyword specifies whether decimal data that is not valid is fixed by the compiler.

You can specify any or all of the data types in any order. However, if a decimal data type is specified, the *NOxxxx parameter for the same data type cannot also be used, and vice versa. For example, if you specify *ZONED you cannot also specify *NOZONED, and vice versa. Separate the parameters with a colon. A parameter cannot be specified more than once.

Note: If the keyword FIXNBR 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 FIXNBR(*NOINPUTPACKED) is specified on the Control specification, then for the pair (*ZONED, *NOZONED), whatever was specified implicitly or explicitly on the command will be used.

If *ZONED is specified, then zoned decimal data that is not valid will be fixed by the compiler on the conversion to packed data. Blanks in numeric fields will be treated as zeros. Each decimal digit will be checked for validity. If a decimal digit is not valid, it is replaced with zero. If a sign is not valid, the sign will be forced to a positive sign code of hex 'F'. If the sign is valid, it will be changed to either a positive sign hex 'F' or a negative sign hex 'D', as appropriate. If the resulting packed data is not valid, it will not be fixed.

If *NOZONED is specified, then zoned decimal data is not fixed by the compiler on the conversion to packed data and will result in decimal errors during runtime if used.

If *INPUTPACKED is specified, then the internal variable will be set to zero if packed decimal data that is not valid is encountered while processing input specifications.

If *NOINPUTPACKED is specified, then decimal errors will occur if packed decimal data that is not valid is encountered while processing input specifications.

If the FIXNBR keyword is not specified, then the values specified on the command are used.

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).

FORMSALIGN{(*NO | *YES)}

The FORMSALIGN keyword indicates that the first line of an output file conditioned with the 1P indicator can be printed repeatedly, allowing you to align the printer. If not specified or specified with *NO, no alignment will be performed. If specified, optionally with *YES, first page forms alignment will occur.

Rules for Forms Alignment

- The records specified on Output Specifications for a file with a device entry for a printer type device conditioned by the first page indicator (1P) may be written as many times as desired. The line will print once. The operator will then have the option to print the line again or continue with the rest of the program.
- All spacing and skipping specified will be performed each time the line is printed.
- When the option to continue with the rest of the program is selected, the line will not be reprinted.

- The function may be performed for all printer files.
- If a page field is specified, it will be incremented only the first time the line is printed.
- When the continue option is selected, the line count will be the same as if the function were performed only once when line counter is specified.

FTRANS{(*NONE | *SRC)}

The FTRANS keyword specifies whether file translation will occur. If specified, optionally with *SRC, file translation will take place and the translate table must be specified in the program. If not specified or specified with *NONE, no file translation will take place and the translate table must not be present.

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.

Note: Any indentation that you request here will not be reflected in the listing debug view that is created when you specify DBGVIEW(*LIST).

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.

LANGID(*JOBRUN | *JOB | 'language-identifier')

The LANGID keyword indicates which language identifier is to be used when the sort sequence is *LANGIDUNQ or *LANGIDSHR. The LANGID keyword is used in conjunction with the SRTSEQ command parameter or keyword to select the sort sequence table.

If *JOBRUN is specified, then the LANGID value associated with the job when the RPG object is executed is used.

If *JOB is specified, then the LANGID value associated with the job when the RPG object is created is used.

A language identifier can be specified, for example, 'FRA' for French and 'DEU' for German.

If the LANGID keyword is not specified, then the value specified on the command is used.

MAIN(main_procedure_name)

The MAIN keyword indicates that this source program is for a linear-main module and contains a linear-main procedure, identified by the *main_procedure_name* parameter, which will be the program entry procedure for the module.

The *main_procedure_name* must be the name of a procedure defined in the source program. The linear-main procedure is intended to be called only through the program call interface and not as a bound procedure call; if you make a recursive call to the linear-main procedure, the call will be a dynamic program call.

Therefore, the following rules apply:
- If a prototype is specified for the linear-main procedure, it must specify the EXTPGM keyword.
- If a prototype is not specified for the linear-main procedure, and a procedure interface is specified, the procedure interface must specify the EXTPGM keyword.
- If the program has no parameters, and the program is not called from an RPG program, neither a prototype nor a procedure interface is required.
- The procedure cannot be exported; the EXPORT keyword may not be specified on the procedure-begin specification for main_procedure_name.

A linear-main module will not include logic for the RPG program cycle; thus language features dependent on the cycle may not be specified.

Note: The NOMAIN keyword also allows you to create a module that does not contain the RPG program cycle.

See “Linear Module” on page 29 for more information.

The following two examples show a linear-main program and its /COPY file.

```
     *  The prototype for the linear-main procedure must have
     *  the EXTPGM keyword with the name of the actual program.
    D DisplayCurTime  PR                 EXTPGM('DSPCURTIME')
```

Figure 108. /COPY file DSPCURTIME used in the following sample linear-main program

```
* The program is named DSPCURTIME, and the module has
      * a linear-main procedure called DisplayCurTime.

      * The Control specification MAIN keyword signifies that this is
      * a linear-main module, and identifies which procedure is the
      * special subprocedure which serves as the linear-main procedure,
      * which will act as the program-entry procedure.

     H MAIN(DisplayCurTime)
      * Copy in the prototype for the program
      /COPY DSPCURTIME
      *--------------------------------------------------
      * Procedure name: DisplayCurTime
      *--------------------------------------------------
     P DisplayCurTime  B
     D DisplayCurTime  PI
      /FREE
       dsply ('It is now ' + %char(%time()));
      /END-FREE
     P DisplayCurTime  E
```

Figure 109. A sample linear-main procedure used in a program

The following example shows a linear main program that does not require a prototype. The program is named PRTCUSTRPT, and the module has a linear-main procedure called PrintCustomerReport. The program is intended to be the command processing program for a *CMD object, so there is no need for an RPG prototype. The Control specification MAIN keyword signifies that this is a linear-main module, and identifies which procedure is the special subprocedure which serves as the linear-main procedure, which will act as the program-entry procedure.

```
     H MAIN(PrintCustomerReport)

      *--------------------------------------------------
      * Program name: PrintCustomerReport (PRTCUSTRPT)
      *--------------------------------------------------

     P PrintCustomerReport...
     P                 B
     F ... file specifications
     D                 PI             EXTPGM('PRTCUSTRPT')
     D  custName                 25A  CONST

       ... calculations, using the custName parameter


     P PrintCustomerReport...
     P                 E
```

Figure 110. A linear main program that is not intended to be called from within any RPG program or procedure

NOMAIN

The NOMAIN keyword indicates that there is no main procedure in this module. It also means that the module in which it is coded cannot be a program-entry module. Consequently, if NOMAIN is specified, then you cannot use the CRTBNDRPG command to create a program. Instead you must either use the CRTPGM command to bind the module with NOMAIN specified to another module that has a program entry procedure or you must use the CRTSRVPGM command.

A no-main module will not include logic for the RPG program cycle; thus language features dependent on the cycle must not be specified.

Note: In addition to the NOMAIN keyword, the MAIN keyword also allows you to create a module that does not contain the RPG program cycle.

See "Linear Module" on page 29 for more information.

OPENOPT (*NOINZOFL | *INZOFL)

For a program that has one or more printer files defined with an overflow indicator (OA-OG or OV), the OPENOPT keyword specifies whether the overflow indicator should be reset to *OFF when the file is opened. If the OPENOPT keyword is specified, with *NOINZOFL, the overflow indicator will remain unchanged when the associated printer file is opened. If not specified or specified with *INZOFL, the overflow indicator will be set to *OFF when the associated printer file is opened.

OPTIMIZE(*NONE | *BASIC | *FULL)

The OPTIMIZE keyword specifies the level of optimization, if any, of the object.

If *NONE is specified, then the generated code is not optimized. This is the fastest in terms of translation time. It allows you to display and modify variables while in debug mode.

If *BASIC is specified, it performs some optimization on the generated code. This allows user variables to be displayed but not modified while the program is in debug mode.

If *FULL is specified, then the most efficient code is generated. Translation time is the longest. In debug mode, user variables may not be modified but may be displayed, although the presented values may not be the current values.

If the OPTIMIZE keyword is not specified, then the value specified on the command is used.

OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP) *{NO}SRCSTMT) *{NO}DEBUGIO) *{NO}UNREF

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 also 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 *SRCSTMT is specified, statement numbers for the listing are generated from the source ID and SEU sequence numbers as follows:

```
stmt_num = source_ID * 1000000 + source_SEU_sequence_number
```

For example, the main source member has a source ID of 0. If the first line in the source file has sequence number 000100, then the statement number for this specification would be 100. A line from a /COPY file member with source ID 27 and source sequence number 000100 would have statement number 27000100. *NOSRCSTMT indicates that line numbers are assigned sequentially.

If *DEBUGIO is specified, breakpoints are generated for all input and output specifications. *NODEBUGIO indicates that no breakpoints are to be generated for these specifications.

If *UNREF is specified, all variables are generated into the module. If *NOUNREF is specified, unreferenced variables are not generated unless they are needed by some other module. The following rules apply to OPTION(*NOUNREF):

- Variables defined with EXPORT are always generated into the module whether or not they are referenced.
- Unreferenced variables defined with IMPORT are generated into the module if they appear on Input specifications.
- The *IN indicator array and the *INxx indicators are not generated into the module if no *IN indicator is used in the program, either explicitly by a *INxx reference, or implicitly by conditioning or result indicator entries.
- For variables not defined with EXPORT or IMPORT:
  - Variables associated with Files, or used in Calculations or on Output specifications are always generated.
  - Variables that appear only on Definition specifications are not generated into the module if they are not referenced.
  - Variables that are referenced only by Input specifications are generated into the module only if DEBUG, DEBUG(*YES) or DEBUG(*INPUT) is specified on the Control specification.

If the OPTION keyword is not specified, then the values specified on the command are used.

PGMINFO(*PCML | *NO { : *MODULE })

The PGMINFO keyword specifies how program-interface information is to be generated for the module or program.

The first parameter specifies whether program-interface information is to be generated. Specifying *PCML indicates that program-interface information is to be generated. Specifying *NO indicates that no program-interface information is to be generated.

The second parameter is not allowed if the first parameter is *NO. Otherwise, the second parameter is required; it must be *MODULE, indicating that program-interface information is to be generated directly into the module. If the module is later used to create a program or service program, the program-interface information will also be place in the program or service program. The information can then be retrieved using API QBNRPII.

The PGMINFO setting defaults to the values specified on the PGMINFO and INFOSTMF parameters of the CRTRPGMOD or CRTBNDRPG command. If the PGMINFO keyword conflicts with the PGMINFO and INFOSTMF command parameters, the value of the Control specification keyword overrides the values specified on the command. However, if the requests from the command parameters and the PGMINFO keyword are different but not in conflict, the compiler will merge the values of the command parameters and the PGMINFO keyword.

Examples

- If the command parameters, for example PGMINFO(*PCML) and INFOSTMF('mypgm.pcml'), specify that the information should be placed in a stream file, and the PGMINFO(*PCML:*MODULE) keyword specifies that the information should be placed in the module, then both requests will be merged, and the final PGMINFO values will be PGMINFO(*PCML:*ALL) INFOSTMF('mypgm.pcml').

- If the command parameters PGMINFO(*PCML *ALL) INFOSTMF('/home/mypcml/mypgm.pcml')specify that the information should be placed both in the module and in a stream file, and the PGMINFO(*NO) keyword specifies that no information should be saved, then the PGMINFO keyword will override the command values, and the final PGMINFO value will be PGMINFO(*NO).

PRFDTA(*NOCOL | *COL)

The PRFDTA keyword specifies whether the collection of profiling data is enabled.

If *NOCOL is specified, the collection of profiling data is not enabled for this object.

If *COL is specified, the collection of profiling is enabled for this object. *COL can be specified only when the optimization level of the object is *FULL.

If the PRFDTA keyword is not specified, then the value specified on the command is used.

SRTSEQ(*HEX | *JOB | *JOBRUN | *LANGIDUNQ | *LANGIDSHR | 'sort-table-name')

The SRTSEQ keyword specifies the sort sequence table that is to be used in the ILE RPG source program.

If *HEX is specified, no sort sequence table is used.

If *JOB is specified, the SRTSEQ value for the job when the *PGM is created is used.

If *JOBRUN is specified, the SRTSEQ value for the job when the *PGM is run is used.

If *LANGIDUNQ is specified, a unique-weight table is used. This special value is used in conjunction with the LANGID command parameter or keyword to determine the proper sort sequence table.

If *LANGIDSHR is specified, a shared-weight table is used. This special value is used in conjunction with the LANGID command parameter or keyword to determine the proper sort sequence table.

A sort table name can be specified to indicate the name of the sort sequence table to be used with the object. It can also be qualified by a library name followed by a slash delimiter ('library-name/sort-table-name'). The library-name is the name of the library to be searched. If a library name is not specified, *LIBL is used to find the sort table name.

If you want to use the SRTSEQ and LANGID parameters to determine the alternate collating sequence, you must also specify ALTSEQ(*EXT) on the control specification.

If the SRTSEQ keyword is not specified, then the value specified on the command is used.

STGMDL(*INHERIT | *SNGLVL | *TERASPACE)

The STGMDL keyword specifies the storage model for the program or module.

- *SNGLVL is used to specify the single-level storage model.
- *INHERIT is used to specify the inherit storage model.
- *TERASPACE is used to specify the teraspace storage model.

When a single-level storage model program or service program is activated and run, it is supplied single-level storage for automatic and static storage. A single-level storage program or service program runs only in a single-level storage activation group. A program compiled with DFTACTGRP(*YES) must be a single-level storage model program.

When a teraspace storage model program or service program is activated and run, it is supplied teraspace storage for automatic and static storage. A teraspace storage program or service program runs only in a teraspace storage activation group.

When an inherit storage model program or service program is activated, it adopts the storage model of the activation group into which it is activated. An equivalent view is that it inherits the storage model of its caller. When the *INHERIT storage model is selected, *CALLER must be specified for the activation group through the ACTGRP parameter or keyword.

An inherit storage model module can be bound into programs and service programs with a storage model of single-level, teraspace or inherit.

A single-level storage model module can only be bound into programs and service programs that use single-level storage.

A teraspace storage model module can only be bound into programs and service programs that use teraspace storage.

If the STGMDL keyword is not specified, then the value specified on the command is used.

TEXT(*SRCMBRTXT | *BLANK | 'description')

The TEXT keyword allows you to enter text that briefly describes the object and its function. The text is used when creating the object and appears when object information is displayed.

If *SRCMBRTXT is specified, the text of the source member is used.

If *BLANK is specified, no text will appear.

If a literal is specified, it can be a maximum of 50 characters and must be enclosed in apostrophes. (The apostrophes are not part of the 50-character string.)

If the TEXT keyword is not specified, then the value specified on the command is used.

THREAD(*CONCURRENT | *SERIALIZE)

The THREAD keyword indicates that the ILE RPG module being created is intended to run safely in a multithreaded environment. One of the major thread-safety issues is the handling of static storage. When multiple threads access the same storage location at the same time, unpredictable results can occur.

Specifying the THREAD keyword helps you make your module thread-safe with regards to the static storage in the module. You can choose between having separate static storage for each thread, or limiting access to the module to only one thread at a time. You can mix the two types of modules in the same program, or service program. However, you should not omit the THREAD keyword in any module that may run in a multithreaded environment.

You do not have to be concerned about automatic variables. Automatic variables are naturally thread-safe because they are created for each invocation of a procedure. Automatic storage for a procedure is allocated in storage which is unique for each thread.

THREAD(*CONCURRENT)

If THREAD(*CONCURRENT) is specified, then multiple threads can run in the module at the same time. By default, all the static storage in the module will be in thread-local storage, meaning that each thread will have its own copy of the static variables in the module, including compiler-internal variables. This allows multiple threads to run the procedures within the module at the same time and be completely independent of each other. For example, one thread could be in the middle of a loop that is reading a file in procedure PROCA, at the same time as another thread is running in an earlier part of PROCA, preparing to open the file for its own use. If the module has a global variable NAME, the value of NAME could be 'Jack' in one thread and 'Jill' in another. The thread-local static variables allow the threads to operate independently.

You can choose to have some of your static variables shared among all threads by using the STATIC(*ALLTHREAD) keyword. If you use this keyword, you are responsible for ensuring that your procedures use that storage in a thread-safe way. See “THREAD(*CONCURRENT | *SERIALIZE)” on page 275.

You can choose to serialize access to individual procedures by specifying the SERIALIZE keyword on the Procedure-Begin specification. If you want to ensure that only one thread is active at one time in a particular part of section of the code, you can move that code to a serialized procedure.

THREAD(*SERIALIZE)

If THREAD(*SERIALIZE) is specified, access to the procedures in the module is serialized. When called in a multithreaded environment, any code within the module can be used by at most one thread at a time.

General thread considerations

To see the advantages and disadvantages of the two types of thread-safety for RPG, see the section on multithreaded applications in *Rational Development Studio for i: ILE RPG Programmer's Guide*. For a list of system functions that are not allowed or supported in a multithreaded environment, see the Multithreaded Applications document under the Programming topic at the following URL:

`http://www.ibm.com/systems/i/infocenter/`

You cannot use the following in a thread-safe program:

- *INUx indicators
- External indicators (*INU1 - *INU8)
- The LR indicator for the CALL or CALLB operation

When using the THREAD keyword, remember the following:

- It is up to the programmer to ensure that storage that is shared across modules or threads is used in a thread-safe manner. This includes:
  - Storage explicitly shared by being exported and imported.
  - Storage shared because a procedure saves the address of a parameter or a pointer parameter, or allocated storage, and uses it on a subsequent call.
  - Storage shared because STATIC(*ALLTHREAD) was specified on the definition of the variable.
- If shared files are used by more than one language (RPG and C, or RPG and COBOL), ensure that only one language is accessing the file at one time.

TIMFMT(fmt{separator})

The TIMFMT keyword specifies the internal time format for time literals and the default internal format for time fields within 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.

If the TIMFMT keyword is not specified the *ISO format is assumed. For more information on internal formats, see “Internal and External Formats” on page 179.

Table 36 on page 209 shows the time formats supported and their separators.

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.

USRPRF(*USER | *OWNER)

The USRPRF keyword specifies the user profile that will run the created program object. The profile of the program owner or the program user is used to run the program and to control which objects can be used by the program (including the authority the program has for each object). This keyword is not updated if the program already exists.

If *USER is specified, the user profile of the program's user will run the created program object.

If *OWNER is specified, the user profiles of both the program's user and owner will run the created program object. The collective set of object authority in both user profiles is used to find and access objects while the program is running. Any objects created during the program are owned by the program's user.

If the USRPRF keyword is not specified, then the value specified on the command is used.

The USRPRF keyword is valid only if the CRTBNDRPG command is used.

Chapter 13. File Description Specifications

File description specifications identify each file used by a module or procedure. 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 RPG program (using input and output specifications). Externally described files have their record and field descriptions defined externally using DDS, DSU, IDDU, or SQL commands. (DSU is part of the CODE/400 product.)

The following limitations apply:

- Only one primary file can be specified. It must be specified as a global file. The presence of a primary file is not required.
- Only one record-address file is a allowed in a module; it must be defined as a global file.
- A maximum of eight PRINTER files is allowed for global files defined in the main source section, and a maximum of eight local PRINTER files is allowed in each procedure.
- There is no limit for the maximum number of files allowed.
- Local files defined in subprocedures must be full-procedural files.
- Files defined in subprocedures do not have Input and Output specifications, so all input and output must be done using data structures.

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-commentary part of the specification extends from position 7 to position 80
  - the fixed-format entries extend from positions 7 to 42. For files defined with the LIKEFILE keyword, the entries from position 17 to position 43 must be blank. The values for those fixed-form entries are taken from the parent file specified by the LIKEFILE keyword.
  - 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++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 111. 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 112. File-Description Keyword Continuation Line Layout

Position 6 (Form Type)

An F must be entered in this position for file description specifications.

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.

At compile time:

- If the file is program-described, the file named in position 7 does not need to exist.
- If the file is externally-described, the file named in position 7 must exist but you can use an IBM i system override command to associate the name to a file defined to the IBM i system, or you can use the EXTDESC keyword to indicate the file defined to the system.

At run time:

- If you use the EXTFILE keyword, the EXTMBR keyword, 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.
- If an IBM i 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 | *EXTDESC)” on page 295 keyword for more information about how overrides interact with this keyword.

When files that are not defined by the USROPN keyword are opened at run time, they are opened in the reverse order to that specified in the file description specifications. The RPG IV 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 if the file is a global primary, secondary, or full procedural file
- 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, if the EXTDESC keyword is not specified, 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 RPG IV functions, such as control fields or 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. If the file is qualified, due to the QUALIFIED or LIKEFILE keywords, the qualified record format is referred to by both the file and record format, for example MYFILE.MYFMT.
- A record format name must be a unique symbolic name. If the file is qualified, due to the QUALIFIED or LIKEFILE keyword, the name of record format must be unique to the other formats of the file. If the file is not qualified, the name of the record format must be unique to the other names within the module.
- RPG IV does not support an externally described logical file with two record formats of the same name. However, such a file can be accessed if it is program described.

Position 17 (File Type)

This entry must be blank if the LIKEFILE keyword is specified. The File Type of the parent file is used.

Entry	Explanation
I	Input file
O	Output file
U	Update file
C	Combined (input/output) file.

Input Files

An input file is one from which a program reads information. It can contain data records, arrays, or tables, or it can be a record-address file.

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 or WORKSTN file. A combined file is also valid for a DISK or SEQ file if position 18 contains T (an array or table replacement file).

Position 18 (File Designation)

This entry must be blank if the LIKEFILE keyword is specified. The File Designation of the parent file is used.

Entry	Explanation
Blank	Output file
P	Primary file
S	Secondary file
R	Record address file
T	Array or table file
F	Full procedural file

You cannot specify P, S, or R if the keyword MAIN or NOMAIN is specified on a control specification.

Primary File

When several files are processed by cycle processing, one must be designated as the primary file. In multi-file processing, processing of the primary file takes precedence. Only one primary file is allowed per program.

Secondary File

When more than one file is processed by the RPG cycle, the additional files are specified as secondary files. Secondary files must be input capable (input, update, or combined file type). The processing of secondary files is determined by the order in which they are specified in the file description specifications and by the rules of multi-file logic.

Record Address File (RAF)

A record-address file is a sequentially organized file used to select records from another file. Only one file in a program can be specified as a record-address file. This file is described on the file description specification and not on the input specifications. A record-address file must be program-described; however, a record-address file can be used to process a program described file or an externally described file.

The file processed by the record-address file must be a primary, secondary, or full-procedural file, and must also be specified as the parameter to the RAFDATA keyword on the file description specification of the record-address file.

You cannot specify a record-address file for the device SPECIAL.

UCS-2 fields are not allowed as the record address type for record address files.

A record-address file that contains relative-record numbers must also have a T specified in position 35 and an F in position 22.

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 or SEQ 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). In addition to a C in position 17, the filename in positions 7-16 must also be specified as the parameter to the TOFILE keyword on the definition specification.

Full Procedural File

A full procedural file is not processed by the RPG cycle: input is controlled by calculation operations. File operation codes such as CHAIN or READ are used to do input functions.

Position 19 (End of File)

Entry	Explanation
E	All records from the file must be processed before the program can end. This entry is not valid for files processed by a record-address file. All records from all files which use this option must be processed before the LR indicator is set on by the RPG cycle to end the program.
Blank	If position 19 is blank for all files, all records from all files must be processed before end of program (LR) can occur. If position 19 is not blank for all files, all records from this file may or may not be processed before end of program occurs in multi-file processing.

Use position 19 to indicate whether the program can end before all records from the file are processed. An E in position 19 applies only to input, update, or combined files specified as primary, secondary, or record-address files.

If the records from all primary and secondary files must be processed, position 19 must be blank for all files or must contain E's for all files. For multiple input files, the end-of-program (LR) condition occurs when all input files for which an E is specified in position 19 have been processed. If position 19 is blank for all files, the end-of-program condition occurs when all input files have been processed.

When match fields are specified for two or more files and an E is specified in position 19 for one or more files, the LR indicator is set on after:

- The end-of-file condition occurs for the last file with an E specified in position 19.
- The program has processed all the records in other files that match the last record processed from the primary file.
- The program has processed the records in those files without match fields up to the next record with non-matching match fields.

When no file or only one file contains match field specifications, no records of other files are processed after end of file occurs on all files for which an E is specified in position 19.

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. This entry must be blank if the LIKEFILE keyword is specified.

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 46 for the relationship between position 17 and position 20 of the file description specifications and positions 18 through 20 of the output specifications.

Table 46. 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
Note: 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 (Sequence)

Entry	**Explanation**
A or blank	Match fields are in ascending sequence.
D	Match fields are in descending sequence.

Position 21 specifies the sequence of input fields used with the match fields specification (positions 65 and 66 of the input specifications). Position 21 applies only to input, update, or combined files used as primary or secondary files. Use positions 65 and 66 of the input specifications to identify the fields containing the sequence information.

If more than one input file with match fields is specified in the program, a sequence entry in position 21 can be used to check the sequence of the match fields and to process the file using the matching record technique. The sequence need only be specified for the first file with match fields specified. If sequence is specified for other files, the sequence specified must be the same; otherwise, the sequence specified for the first file is assumed.

If only one input file with match fields is specified in the program, a sequence entry in position 21 can be used to check fields of that file to ensure that the file is in sequence. By entering one of the codes M1 through M9 in positions 65 and 66 of the input specifications, and by entering an A, blank, or D in position 21, you specify sequence checking of these fields.

Sequence checking is required when match fields are used in the records from the file. When a record from a matching input file is found to be out of sequence, the RPG IV exception/error handling routine is given control.

Position 22 (File Format)

This entry must be blank if the LIKEFILE keyword is specified. The File Format of the parent file is used.

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 and record-address files).

An E in position 22 indicates that the record descriptions for the file are external to the RPG IV source program. The compiler obtains these descriptions at compilation time and includes them in the source program.

Positions 23-27 (Record Length)

This entry must be blank if the LIKEFILE keyword is specified. The Record Length of the parent file is used.

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. This entry must be blank for externally described files.

If the file being defined is a record-address file and the record length specified is 3, it is assumed that each record in the file consists of a 3-byte binary field for the relative-record numbers starting at offset 0. If the record length is 4 or greater, each relative-record number in the record-address file is assumed to be a 4-byte field starting at offset 1. If the record length is left blank, the actual record length is retrieved at run time to determine how to handle the record-address file.

If the file opened at run time has a primary record length of 3, then 3-byte relative-record numbers (one per record) are assumed; otherwise, 4-byte relative-record numbers are assumed. This support can be used to allow ILE RPG programs to use System/36™ environment SORT files as record-address files.

Table 47. Valid Combinations for a Record Address File containing Relative Record Numbers (RAFRRN)

Record Length Positions 23-27	RAF Length Positions 29-33	Type of Support
Blank	Blank	Support determined at run time.
3	3	System/36 support.
> = 4	4	Native support.

Position 28 (Limits Processing)

Entry **Explanation**

L Sequential-within-limits processing by a record-address file

Blank Sequential or random processing

Use position 28 to indicate whether the file is processed by a record-address file that contains limits records.

A record-address file used for limits processing contains records that consist of upper and lower limits. Each record contains a set of limits that consists of the lowest record key and the highest record key from the segment of the file to be processed. Limits processing can be used for keyed files specified as primary, secondary, or full procedural files.

The L entry in position 28 is valid only if the file is processed by a record-address file containing limits records. Random and sequential processing of files is implied by a combination of positions 18 and 34 of the file description specifications, and by the calculation operation specified.

The operation codes "SETLL (Set Lower Limit)" on page 808 and "SETGT (Set Greater Than)" on page 804 can be used to position a file; however, the use of these operation codes does not require an L in this position.

For more information on limits processing, refer to the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Positions 29-33 (Length of Key or Record Address)

This entry must be blank if the LIKEFILE keyword is specified. The Length of Key of the parent file is used.

Entry	Explanation
1-2000	The number of positions required for the key field in a program described file or the length of the entries in the record-address file (which must be a program-described file). If the program-described file being defined uses keys for record identification, enter the number of positions occupied by each record key. This entry is required for indexed files. If the keys are packed, the key field length should be the packed length; this is the number of digits in DDS divided by 2 plus 1 and ignoring any fractions. If the file being defined is a record-address file, enter the number of positions that each entry in the record-address file occupies. If the keys are graphic, the key field length should be specified in bytes (for example, 3 graphic characters requires 6 bytes).
Blank	These positions must be blank for externally described files. (The key length is specified in the external description.) For a program-described file, a blank entry indicates that keys are not used. Positions 29-33 can also be blank for a record-address file with a blank in positions 23-27 (record length).

Position 34 (Record Address Type)

This entry must be blank if the LIKEFILE keyword is specified. The Record Address Type of the parent file is used.

Entry	Explanation
Blank	Relative record numbers are used to process the file. Records are read consecutively. Record address file contains relative-record numbers. For limits processing, the record-address type (position 34) is the same as the type of the file being processed.
A	Character keys (valid only for program-described files specified as indexed files or as a record-address-limits file).
P	Packed keys (valid only for program-described files specified as indexed files or as a record-address-limits file).
G	Graphic keys (valid only for program-described files specified as indexed files or as a record-address-limits file).
K	Key values are used to process the file. This entry is valid only for externally described files.
D	Date keys are used to process the file. This entry is valid only for program-described files specified as indexed files or as a record-address-limits file.
T	Time keys are used to process the file. This entry is valid only for program-described files specified as indexed files or as a record-address-limits file.
Z	Timestamp Keys are used to process the file. This entry is valid only for program-described files specified as indexed files or as a record-address-limits file.
F	Float Key (valid only for program-described files specified as indexed files or as a record-address-limits file).

UCS-2 fields are not allowed as the record address type for program described indexed files or record address files.

Blank=Non-keyed Processing

A blank indicates that the file is processed without the use of keys, that the record-address file contains relative-record numbers (a T in position 35), or that the keys in a record-address-limits file are in the same format as the keys in the file being processed.

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.

A=Character Keys

The indexed file (I in position 35) defined on this line is processed by character-record keys. (A numeric field used as the search argument is converted to zoned decimal before chaining.) The A entry must agree with the data format of the field identified as the key field (length in positions 29 to 33 and starting position specified as the parameter to the KEYLOC keyword).

The record-address-limits file (R in position 18) defined on this line contains character keys. The file being processed by this record address file can have an A, P, or K in position 34.

P=Packed Keys

The indexed file (I in position 35) defined on this line is processed by packed-decimal-numeric keys. The P entry must agree with the data format of the field identified as the key field (length in positions 29 to 33 and starting position specified as the parameter to the KEYLOC keyword).

The record-address-limits file defined on this line contains record keys in packed decimal format. The file being processed by this record address file can have an A, P, or K in position 34.

G=Graphic Keys

The indexed file (I in position 35) defined on this line is processed by graphic keys. Since each graphic character requires two bytes, the key length must be an even number. The record-address file which is used to process this indexed file must also have a 'G' specified in position 34 of its file description specification, and its key length must also be the same as the indexed file's key length (positions 29-33).

K=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.

D=Date Keys

The indexed file (I in position 35) defined on this line is processed by date keys. The D entry must agree with the data format of the field identified as the key field (length in positions 29 to 33 and starting position specified as the parameter to the KEYLOC keyword).

The hierarchy used when determining the format and separator for the date key is:

1. From the DATFMT keyword specified on the file description specification
2. From the DATFMT keyword specified in the control specification
3. *ISO

T=Time Keys

The indexed file (I in position 35) defined on this line is processed by time keys. The T entry must agree with the data format of the field identified as the key field (length in positions 29 to 33 and starting position specified as the parameter to the KEYLOC keyword).

The hierarchy used when determining the format and separator for the time key is:

1. From the TIMFMT keyword specified on the file description specification
2. From the TIMFMT keyword specified in the control specification
3. *ISO

Z=Timestamp Keys

The indexed file (I in position 35) defined on this line is processed by timestamp keys. The Z entry must agree with the data format of the field identified as the key field (length in positions 29 to 33 and starting position specified as the parameter to the KEYLOC keyword).

F=Float Keys

The indexed file (I in position 35) defined on this line is processed by float keys. The Length-of-Key entry (positions 29-33) must contain a value of either 4 or 8 for a float key. When a file contains a float key, any type of numeric variable or literal may be specified as a key on keyed input/output operations. For a non-float record address type, you cannot have a float search argument.

For more information on record address type, refer to the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Position 35 (File Organization)

This entry must be blank if the LIKEFILE keyword is specified. The File Organization of the parent file is used.

Entry	Explanation
Blank	The program-described file is processed without keys, or the file is externally described.
I	Indexed file (valid only for program-described files).
T	Record address file that contains relative-record numbers (valid only for program-described files).

Use position 35 to identify the organization of program described files.

Blank=Non-keyed Program-Described File

A program-described file that is processed without keys can be processed:
- Randomly by relative-record numbers, positions 28 and 34 must be blank.
- Entry Sequence, positions 28 and 34 must be blank.
- As a record-address file, position 28 must be blank.

I=Indexed File

An indexed file can be processed:
- Randomly or sequentially by key
- By a record-address file (sequentially within limits). Position 28 must contain an L.

T=Record Address File

A record-address file (indicated by an R in position 18) that contains relative-record numbers must be identified by a T in position 35. (A record-address file must be program described.) Each record retrieved from the file being processed is based on the relative record number in the record-address file. (Relative record numbers cannot be used for a record-address-limits file.)

Each relative-record number in the record-address file is a 4-byte binary field; therefore, each 4-byte unit of a record-address file contains a relative-record number. A minus one (-1 or hexadecimal FFFFFFFF) relative-record number value causes the record to be skipped. End of file occurs when all record-address file records have been processed.

For more information on how to handle record-address files, see the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Positions 36-42 (Device)

This entry must be blank if the LIKEFILE keyword is specified. The Device entry of the parent file is used.

Entry	Explanation
PRINTER	File is a printer file, a file with control characters that can be sent to a printer.
DISK	File is a disk file. This device supports sequential and random read/write functions. These files can be accessed on a remote system by Distributed Data Management (DDM).
WORKSTN	File is a workstation file. Input/output is through a display or ICF file.
SPECIAL	This is a special file. Input or output is on a device that is accessed by a user-supplied program. The name of the program must be specified as the parameter for the PGMNAME 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 304 and "PGMNAME(program_name)" on page 304 for more information.
SEQ	File is a sequentially organized file. The actual device is specified in a CL command or in the file description, which is accessed by the file name.

Use positions 36 through 42 to specify the RPG IV device name to be associated with the file. The RPG IV device name defines the ILE RPG functions that can be done on the associated file. Certain functions are valid only for a specific ILE RPG device name (such as the EXFMT operation for WORKSTN). The file name specified in positions 7 through 16 can be overridden at run time, allowing you to change the input/output device used in the program.

Note that the RPG IV device names are not the same as the system device names.

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 Keywords

File-Description 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 file-description keywords, the keyword field can be continued on subsequent lines. See “File-Description Keyword Continuation Line” on page 279 and “File Description Specification Keyword Field” on page 251.

ALIAS

When the ALIAS keyword is specified for an externally-described file, the RPG compiler will use the alias (alternate) names, if present, when determining the subfield names for data structures defined with the LIKEREC keyword. When the ALIAS keyword is not specified for the RPG file, or an external field does not have an alias name defined, the RPG compiler will use the standard external field name.

Note: If the alternate name for a particular external field is enclosed in quotes, the standard external field name is used for that field.

The ALIAS keyword is allowed for an externally-described file for which the RPG compiler will not generate Input or Output specifications. This includes files defined with the TEMPLATE or QUALIFIED keyword, and local files defined in subprocedures.

When the PREFIX keyword is specified with the ALIAS keyword, the second parameter of PREFIX, indicating the number of characters to be replaced, does not apply to the alias names. In the following discussion, assume that the file MYFILE has fields XYCUSTNM and XYID_NUM, and the XYCUSTNM field has the alias name CUSTOMER_NAME.
- If keyword PREFIX(NEW_) is specified, there is no second parameter, so no characters are replaced for any names. The names used for LIKEREC subfields will be NEW_CUSTOMER_NAME and NEW_XYID_NUM.
- If keyword PREFIX(NEW_:2) is specified, two characters will be replaced in the names of fields that do not have an alias name. The names used for LIKEREC subfields will be NEW_CUSTOMER_NAME and NEW_ID_NUM. The first two characters, "XY", are replaced in XYID_NUM, but no characters are replaced in CUSTOMER_NAME.
- If keyword PREFIX('':2) is specified, two characters will be repaced in the names of fields that do not have an alias name. The names used for LIKEREC subfields

will be CUSTOMER_NAME and ID_NUM. The first two characters, "XY", are replaced in XYID_NUM, but no characters are replaced in CUSTOMER_NAME.

- If the first parameter for PREFIX contains a data structure name, for example PREFIX('MYDS.'), the part of the prefix before the dot will be ignored.

```
 * The DDS specifications for file MYFILE, using the ALIAS keyword
 * for the first field to associate the alias name CUSTOMER_NAME
 * with the CUSTNM field
A          R CUSTREC
A            CUSTNM        25A         ALIAS(CUSTOMER_NAME)
A            ID_NUM        12P 0

 * The RPG source, using the ALIAS keyword:
Fmyfile    if   e             disk    ALIAS QUALIFIED
 * The subfields of the LIKEREC data structure are
 *    CUSTOMER_NAME (using the ALIAS name)
 *    ID_NUM        (using the standard name)
D myDs            ds                  LIKEREC(myfile.custRec)
 /free
     read myfile myDs;
     if myDs.customer_name <> *blanks
     and myDs.id_num > 0;
       ...
```

Figure 113. Using the ALIAS keyword for an externally-described file

BLOCK(*YES |*NO)

The BLOCK keyword controls the blocking of records associated with the file. The keyword is valid only for DISK or SEQ files.

If this keyword is not specified, the RPG compiler unblocks input records and blocks output records to improve run-time performance in SEQ or DISK files when the following conditions are met:

1. The file is program-described or, if externally described, it has only one record format.
2. Keyword RECNO is not used in the file description specification.

   Note: If RECNO is used, the ILE RPG compiler will not allow record blocking. However, if the file is an input file and RECNO is used, Data Management may still block records if fast sequential access is set. This means that updated records might not be seen right away.
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; it is not a record-address file or processed by a record-address 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.)

If BLOCK(*YES) is specified, record blocking occurs as described above except that the operations SETLL, SETGT, 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. Then no record blocking is done by the compiler.

COMMIT{(rpg_name)}

The COMMIT keyword allows the processing of 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 by RPG to '0'.

By specifying the optional parameter, you can control at run time whether to enable commitment control. If the parameter contains a '1', the file will be opened with the COMMIT indication on, otherwise the file will be opened without COMMIT. The parameter must be set prior to opening the file. If the file is opened at program initialization, the COMMIT parameter can be passed as a call parameter or defined as an external indicator. If the file is opened explicitly, using the OPEN operation in the calculation specifications, the parameter 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.

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 on which this keyword is specified is indexed and the key field is a date, then this also provides the default external format for the key field.

For a Record-Address file this specifies the external date format of date limits keys read from the record-address file.

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 (positions 53-57).

See Table 33 on page 207 for valid formats and separators. For more information on external formats, see "Internal and External Formats" on page 179.

DEVID(fieldname)

The DEVID keyword specifies the name of the program device that supplied the record processed in the file. The field is updated each time a record is read from a file. Also, you may move a program device name into this field to direct an output or device-specific input operation (other than a READ-by-file-name or an implicit cycle read) to a different device.

The fieldname is implicitly defined as a 10-character alphanumeric field. The device name specified in the field must be left-justified and padded with blanks.

Initially, the field is blank. A blank field indicates the requester device. If the requester device is not acquired for your file, you must not use a blank field.

The DEVID field is maintained for each call to a program. If you call program B from within program A, the DEVID field for program A is not affected. Program B uses a separate DEVID field. When you return to program A, its DEVID field has the same value as it had before you called program B. If program B needs to know which devices are acquired to program A, program A must pass this information (as a parameter list) when it calls program B.

If the DEVID keyword is specified but not the MAXDEV keyword, the program assumes a multiple device file (MAXDEV with a parameter of *FILE).

To determine the name of the requester device, you may look in the appropriate area of the file information data structure (see "File Information Data Structure" on page 79). Or, you may process an input or output operation where the fieldname contains blanks. After the operation, the fieldname has the name of the requester device.

EXTDESC(external-filename)

The EXTDESC keyword can be specified to indicate which file the compiler should use at compile time to obtain the external descriptions for the file.

The file specified by the EXTDESC keyword is used only at compile time. At runtime, the file is found using the same rules as would be applied if the EXTDESC keyword was not specified. You can use additional keyword EXTFILE(*EXTDESC) if you also want the file specified by the EXTDESC keyword to be used at runtime.

The EXTDESC keyword must be specified before any keywords that have record format names as parameters such as IGNORE, INCLUDE, RENAME, and SFILE, and before any keywords whose validity depends on the actual file, such as INDDS and SLN.

The parameter for EXTDESC must be a literal specifying a valid file name. 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 EXTDESC('qtemp/myfile'), the file will not be found. Instead, you should specify EXTDESC('QTEMP/MYFILE').
4. If you have specified an override for the file that RPG will use for the external descriptions, that override will be in effect. If the EXTDESC('MYLIB/MYFILE') is specified, RPG will use the file MYLIB/MYFILE for the external descriptions. If the command OVRDBF MYFILE OTHERLIB/OTHERFILE has been used before compiling, the actual file used will be OTHERLIB/OTHERFILE. Note that any overrides for the name specified in positions 7-15 will be ignored, since that name is only used internally within the RPG source member.

```
 * At compile time, file MYLIB/MYFILE1 will be used to
 * get the definition for file "FILE1", as specified by
 * the EXTDESC keyword.
 * At runtime, file *LIBL/FILE1 will be opened.  Since
 * the EXTFILE keyword is not specified, the file name
 * defaults to the RPG name for the file.
Ffile1     if   e             disk
F                             extdesc('MYLIB/MYFILE1')
 * At compile time, file MYLIB/MYFILE2 will be used to
 * get the definition for file "FILE2", as specified by
 * the EXTDESC keyword.
 * At runtime, file MYLIB/MYFILE2 will be opened, as
 * specified by the EXTFILE(*EXTDESC) keyword.
Ffile2     if   e             disk
F                             extdesc('MYLIB/MYFILE2')
F                             extfile(*extdesc)
```

Figure 114. Example of the EXTDESC keyword.

EXTFILE(filename | *EXTDESC)

The EXTFILE keyword specifies which file, in which library, is opened.

filename can be a literal or a variable. You can specify the value in any of the following forms:

```
filename
libname/filename
*LIBL/filename
```

Special value *EXTDESC can be used to indicate that the parameter for the EXTDESC keyword should also be used for the EXTFILE keyword.

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. Use the EXTDESC keyword to locate the file at compile-time.
5. When EXTFILE(*EXTDESC) is specified, the EXTDESC keyword must also be specified for the file, or for the parent file if the file is defined with the LIKEFILE keyword.
6. If a variable name is used, it must be set before the file is opened. For files that are opened automatically during the initialization part of the cycle, 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
   - Using a program-global variable that is set by another module.

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   f   10        disk    extfile(filename)
```

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.

```
 * The name of the file is known at compile time
Ffile1     IF   F   10        DISK    EXTFILE('MYLIB/FILE1')
Ffile2     IF   F   10        DISK    EXTFILE('FILE2')

 * The name of the file is in a variable which is
 * in the correct form when the program starts.
 * Variable "filename3" must have a value such as
 * 'MYLIB/MYFILE' or 'MYFILE' when the file is
 * opened during the initialization phase of the
 * RPG program.
Ffile3     IF   F   10        DISK    EXTFILE(filename3)

 * The library and file names are in two separate variables
 * The USROPN keyword must be used, so that the "filename4"
 * variable can be set correctly before the file is opened.
Ffile4     IF   F   10        DISK    EXTFILE(filename4)
F                                     USROPN
D filename4       S             21A
 * EXTFILE variable "filename4" is set to the concatenated
 * values of the "libnam" and "filnam" variables, to form
 * a value in the form "LIBRARY/FILE".
C                   EVAL      filename4 = %trim(libnam) + '/' + filnam
C                   OPEN      file4

 * At compile time, file MYLIB/MYFILE5 will be used to
 * get the external definition for the file "file5",
 * due to the EXTDESC keyword.
 * At runtime, the file MYLIB/MYFILE5 will be opened,
 * due to the EXTFILE(*EXTDESC) keyword.
Ffile5     if   e             DISK
F                             EXTFILE(*EXTDESC)
F                             EXTDESC('MYLIB/MYFILE5')
```

Figure 115. Examples of the EXTFILE keyword

EXTIND(*INUx)

The EXTIND keyword indicates whether the file is used in the program depending on the value of the external indicator.

EXTIND lets the programmer control the operation of input, output, update, and combined files at run time. If the specified indicator is on at program initialization, the file is opened. If the indicator is not on, the file is not opened and is ignored during processing. The *INU1 through *INU8 indicators can be set as follows:

- By the IBM i control language.
- When used as a resulting indicator for a calculation operation or as field indicators on the input specifications. Setting the *INU1 through *INU8 indicators in this manner has no effect on file conditioning.

  See also “USROPN” on page 312.

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 the initialization part of the cycle, 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
- Using a program-global variable that is set by another module.

FORMLEN(number)

The FORMLEN keyword specifies 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.

Changing the form length does not require recompiling the program. You can override the number parameter of FORMLEN by specifying a new value for the PAGSIZE parameter of the Override With Printer File (OVRPRTF) command.

When the FORMLEN keyword is specified, the FORMOFL keyword must also be specified.

FORMOFL(number)

The FORMOFL keyword specifies the overflow line number that will set on the overflow indicator. The overflow line number must be less than or equal to the form length. When the line that is specified as the overflow line is printed, the overflow indicator is set on.

Changing the overflow line does not require recompiling the program. You can override the number parameter of FORMOFL by specifying a new value for the OVRFLW parameter of the Override With Printer File (OVRPRTF) command.

When the FORMOFL keyword is specified, the FORMLEN keyword must also be specified.

IGNORE(recformat{:recformat...})

The IGNORE keyword allows a record format from an externally described file to be ignored. 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.

Remember that for a qualified file, the unqualified form of the record format name is used for the IGNORE keyword.

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. For WORKSTN files, the record formats specified using the SFILE keyword are also included in the program, they need not be specified twice. Multiple record formats can be specified, separated by colons (:).

When the INCLUDE keyword is specified for a file, the IGNORE keyword cannot be specified.

Remember that for a qualified file, the unqualified form of the record format name is used for the INCLUDE keyword.

INDDS(data_structure_name)

The INDDS keyword lets you associate a data structure name with the INDARA indicators for a workstation or printer file. This data structure contains the conditioning and response indicators passed to and from data management for the file, and is called an indicator data structure.

Rules:

- This keyword is allowed only for externally described PRINTER files and externally and program-described WORKSTN files.
- For a program-described file, the PASS(*NOIND) keyword must not be specified with the INDDS keyword.
- The same data structure name may be associated with more than one file.
- The data structure name must be defined as a data structure on the definition specifications and can be a multiple-occurrence data structure.
- The length of the indicator data structure is always 99.
- The indicator data structure is initialized by default to all zeros ('0's).
- The SAVEIND keyword cannot be specified with this keyword.

If this keyword is not specified, the *IN array is used to communicate indicator values for all files defined with the DDS keyword INDARA.

For additional information on indicator data structures, see “Special Data Structures” on page 141.

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 must be coded in the same scope as the file; for a global file, it must be coded in the main source section, and for a local file, it must be coded in the same subprocedure as the file. Furthermore, it must have the same storage type, static or automatic, as the file.

For additional information on file information data structures, see “File Information Data Structure” on page 79.

INFSR(SUBRname)

The INFSR keyword identifies the file exception/error subroutine that 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 for a global file that is accessed by a subprocedure. The INFSR subroutine must be coded in the same scope as the file; for a local file, it must be coded in the same subprocedure as the file, and for a global file in a cycle module, it must be coded in the main source section.

KEYLOC(number)

The KEYLOC keyword specifies the record position in which the key field for a program-described indexed-file begins. The parameter must be between 1 and 32766.

The key field of a record contains the information that identifies the record. The key field must be in the same location in all records in the file.

LIKEFILE(parent-filename)

The LIKEFILE keyword is used to define one file like another file.

Note: In the following discussion, the term new file is used for the file defined using the LIKEFILE keyword, and the term parent file is used for the parameter of the LIKEFILE keyword whose definition is used to derive the definition of the new file.

Rules for the LIKEFILE keyword:

- When a file is defined with the LIKEFILE keyword, the QUALIFIED keyword is assumed. Record formats are automatically qualified for a file defined with the LIKEFILE keyword. If the record formats of the parent file FILE1 are RECA and RECB, then the record formats of the new file FILE2 must be referred to in the RPG program by FILE2.RECA and FILE2.RECB.
- The QUALIFIED keyword cannot be specified with the LIKEFILE keyword.
- All non-ignored record formats from the parent file are available for the new file.
- If the LIKEFILE keyword is specified, the file specified as a parameter must have already been defined in the source file.
- If the LIKEFILE keyword is specified in a subprocedure, and the file specified as the parameter is defined in the global definitions, the compiler will locate the global definition at the time of scanning the LIKEFILE definition.
- Input and output specifications are not generated or allowed for files defined with LIKEFILE. All input and output operations must be done with result data structures.
- When a file is defined with LIKEFILE, the File specifications for the parent file must make it clear whether or not the file is blocked. It may be necessary to specify the BLOCK keyword for the parent file. For example, for an input DISK file, the BLOCK keyword is required if the file is used in a LIKEFILE keyword since the file is blocked depending on which calculation operations are used for the file. For an Input-Add DISK file, the file can never be blocked, so the BLOCK keyword is not required.
- If BLOCK(*YES) is specified for a file, and the file is used as a parent file for files defined with the LIKEFILE keyword, the READE, READPE and READP

operations are not allowed for the parent file, or for any files related to the parent file through the LIKEFILE keyword.

- Some properties of the parent file are inherited by the new file, and some are not. Of the properties which are inherited, some can be overridden by File specification keywords. The properties which are not inherited can be specified for the new file by File specification keywords, see Table 48.

Table 48. File properties which are inherited and which can be overridden

Property or keyword	**Inherited from parent file**	**Can be specified for new file**
File type (Input, update, output, combined)	Yes	No
File addition	Yes	No
Record address type (RRN, keyed)	Yes	No
Record length (Program-described files)	Yes	No
Key length (Program-described files)	Yes	No
File organization (Program-described files)	Yes	No
Device	Yes	No
BLOCK	Yes	No
COMMIT	No	Yes
DATFMT	N/A, see Note 1	
DEVID	No	Yes
EXTDESC	Yes	No
EXTFILE	Yes, see Note 2	Yes
EXTIND	No	Yes
EXTMBR	Yes, see Note 2	Yes
FORMLEN	Yes	Yes
FORMOFL	Yes	Yes
IGNORE	Yes	No
INCLUDE	Yes	No
INDDS	No	Yes
INFDS	No	Yes
INFSR	No	Yes
KEYLOC	Yes	No
LIKEFILE	Yes	N/A
MAXDEV	Yes	Yes
OFLIND	No	Yes
PASS	Yes	No
PGMNAME	Yes	Yes
PLIST	No	Yes
PREFIX	Yes	No
PRTCTL	No	Yes

Table 48. File properties which are inherited and which can be overridden (continued)

Property or keyword	Inherited from parent file	Can be specified for new file
QUALIFIED	N/A, QUALIFIED is always implied for new file	
RAFDATA	N/A, see Note 3	
RECNO	No	Yes
RENAME	Yes	No
SAVEDS	No	Yes
SAVEIND	No	Yes
SFILE	Yes, see Note 4	Yes, see Note 4
SLN	No	Yes
STATIC	No	Yes
TEMPLATE	No	Yes
TIMFMT	N/A, see Note 1	
USROPN	No	Yes

Notes:

1. The DATFMT and TIMFMT keywords relate to Date and Time fields coded on program-described Input specifications for the file, but Input specifications are not relevant for files defined with the LIKEFILE keyword.
2. The external file associated with the RPG file depends on the EXTFILE and EXTMBR keywords specified for both the parent file and the new file. By default, the external file associated with each file is the name specified in the Name entry for the file. The new file inherits the EXTFILE or EXTMBR keywords from the parent file if the parameters are constants, but these keywords may also be specified for the new file. If the parameter for EXTFILE or EXTMBR is not a constant, the EXTFILE or EXTMBR keyword is not inherited. The following table shows the external files that would be used at runtime for some examples of EXTFILE and EXTMBR values for a parent file and a new file that is defined LIKEFILE the parent file.

Table 49. File specification examples: EXTFILE and EXTMBR

File Specifications	External files used at runtime (Inherited values appear in bold)
Examples where the EXTFILE and EXTMBR values are both constants	
FFILE1 IF E DISK FFILE2 LIKEFILE(FILE1)	*LIBL/FILE1(*FIRST) *LIBL/FILE2(*FIRST)
FFILE1 IF E DISK **EXTFILE('MYLIB/MYFILE')** FFILE2 LIKEFILE(FILE1)	MYLIB/MYFILE(*FIRST) **MYLIB/MYFILE**(*FIRST)
FFILE1 IF E DISK FFILE2 LIKEFILE(FILE1) **EXTFILE('MYLIB/MYFILE')**	*LIBL/FILE1(*FIRST) MYLIB/MYFILE(*FIRST)
FFILE1 IF E DISK **EXTFILE('MYLIB/MYFILE1')** FFILE2 LIKEFILE(FILE1) **EXTFILE('MYLIB/MYFILE2')**	MYLIB/MYFILE1(*FIRST) MYLIB/MYFILE2(*FIRST)
FFILE1 IF E DISK **EXTMBR('MBR1')** FFILE2 LIKEFILE(FILE1)	*LIBL/FILE1(MBR1) *LIBL/FILE2**(MBR1)**
FFILE1 IF E DISK FFILE2 LIKEFILE(FILE1) **EXTMBR('MBR1')**	*LIBL/FILE1(*FIRST) *LIBL/FILE2(MBR1)
FFILE1 IF E DISK EXTMBR('MBR1') FFILE2 LIKEFILE(FILE1) **EXTFILE('MYLIB/MYFILE2')**	*LIBL/FILE1(MBR1) MYLIB/MYFILE2**(MBR1)**

Table 49. File specification examples: EXTFILE and EXTMBR (continued)

File Specifications	External files used at runtime (Inherited values appear in bold)
Examples where the EXTFILE and EXTMBR values are both variable	

`FFILE1  IF  E    DISK` **`EXTFILE(extfileVariable)`**
`FFILE2                LIKEFILE(FILE1)`
`Value of extfileVariable: 'MYLIB/MYFILE'`
	`MYLIB/MYFILE(*FIRST)`
	`*LIBL/FILE2(*FIRST)`

`FFILE1  IF  E    DISK`
`FFILE2                LIKEFILE(FILE1)` **`EXTFILE(extfileVariable)`**
`Value of extfileVariable: 'MYLIB/MYFILE'`
	`*LIBL/FILE1(*FIRST)`
	`MYLIB/MYFILE(*FIRST)`

`FFILE1  IF  E    DISK` **`EXTFILE(extfileVariable1)`**
`FFILE2                LIKEFILE(FILE1)` **`EXTFILE(extfileVariable2)`**
`Value of extfileVariable1: 'MYLIB/MYFILE1'`
`Value of extfileVariable2: 'MYLIB/MYFILE2'`
	`MYLIB/MYFILE1(*FIRST)`
	`MYLIB/MYFILE2(*FIRST)`

`FFILE1  IF  E    DISK` **`EXTMBR(extmbrVariable)`**
`FFILE2                LIKEFILE(FILE1)`
`Value of extmbrVariable: 'MBR1'`
	`*LIBL/FILE1(MBR1)`
	`*LIBL/FILE2(*FIRST)`

`FFILE1  IF  E    DISK`
`FFILE2                LIKEFILE(FILE1)` **`EXTMBR(extmbrVariable)`**
`Value of extmbrVariable: 'MBR1'`
	`*LIBL/FILE1(*FIRST)`
	`*LIBL/FILE2(MBR1)`

`FFILE1  IF  E    DISK` **`EXTMBR(extmbrVariable)`**
`FFILE2                LIKEFILE(FILE1)` **`EXTFILE(extfileVariable)`**
`Value of extmbrVariable: 'MBR1'`
`Value of extfileVariable: 'MYLIB/MYFILE2'`
	`*LIBL/FILE1(MBR1)`
	`MYLIB/MYFILE2(*FIRST)`

Examples where the EXTFILE and EXTMBR values are mixed variables and constants	

`FFILE1  IF  E    DISK` **`EXTFILE(extfileVariable1) EXTMBR('MBR1')`**
`FFILE2                LIKEFILE(FILE1)`
`Value of extfileVariable1: 'MYLIB/MYFILE1'`
	`MYLIB/MYFILE1(MBR1)`
	`*LIBL/FILE2`**`(MBR1)`**

`FFILE1  IF  E    DISK` **`EXTMBR(extmbrVariable)`**
`FFILE2                LIKEFILE(FILE1)`
`Value of extmbrVariable: 'MBR1'`
	`*LIBL/FILE1(MBR1)`
	`*LIBL/FILE2(*FIRST)`

`FFILE1  IF  E    DISK` **`EXTFILE('MYLIB/MYFILE1') EXTMBR(extmbrVariable)`**
`FFILE2                LIKEFILE(FILE1)`
`Value of extmbrVariable: 'MBR1'`
	`MYLIB/MYFILE1(MBR1)`
	`MYLIB/MYFILE1``(*FIRST)`

3. The RAFDATA keyword is relevant only for Primary and Secondary files, but the parent file must be a Full Procedural file.
4. The SFILE keyword indicates that the record format is a subfile record format, and it also indicates the name of the variable used to specify the relative record number for the subfile. The new file automatically inherits the fact that a particular record format is a subfile record format; however, it does not inherit the name of the variable used to specify the RRN. The SFILE keyword must be specified for the new file to indicate which variable is to be used to specify the relative record number for the subfile.

MAXDEV(*ONLY | *FILE)

The MAXDEV keyword specifies the maximum number of devices defined for the WORKSTN file. The default, *ONLY, indicates a single device file. If *FILE is specified, the maximum number of devices (defined for the WORKSTN file on the create-file command) is retrieved at file open, and SAVEIND and SAVEDS space allocation will be done at run time.

With a shared file, the MAXDEV value is not used to restrict the number of acquired devices.

When you specify DEVID, SAVEIND, or SAVEDS but not MAXDEV, the program assumes the default of a multiple device file (MAXDEV with a parameter of *FILE).

OFLIND(indicator)

The OFLIND keyword specifies an overflow indicator to condition which lines in the PRINTER file will be printed when overflow occurs. This entry is valid only for a PRINTER device. Default overflow processing (that is, automatic page eject at overflow) is done if the OFLIND keyword is not specified.

Valid Parameters:

***INOA-*INOG, *INOV:**
Specified overflow indicator conditions the lines to be printed when overflow occurs on a program described printer file.

***IN01-*IN99:**
Set on when a line is printed on the overflow line, or the overflow line is reached or passed during a space or skip operation.

name: The name of a variable that is defined with type indicator and is not an array. This indicator is set on when the overflow line is reached and the program must handle the overflow condition.

The behavior is the same as for indicators *IN01 to *IN99.

Note: Indicators *INOA through *INOG, and *INOV are not valid for externally described files.

Only one overflow indicator can be assigned to a file. If more than one PRINTER file in a module is assigned an overflow indicator, that indicator must be unique for each file. A global indicator cannot be used on more than one file even if one of the files is defined in a different procedure.

PASS(*NOIND)

The PASS keyword determines whether indicators are passed under programmer control or based on the DDS keyword INDARA. This keyword can only be specified for program-described files. To indicate that you are taking responsibility for passing indicators on input and output, specify PASS(*NOIND) on the file description specification of the corresponding program-described WORKSTN file.

When PASS(*NOIND) is specified, the ILE RPG compiler does not pass indicators to data management on output, nor does it receive them on input. Instead you pass indicators by describing them as fields (in the form *INxx, *IN(xx), or *IN) in the input or output record. They must be specified in the sequence required by the data description specifications (DDS). You can use the DDS listing to determine this sequence.

If this keyword is not specified, the compiler assumes that INDARA was specified in the DDS.

Note: If the file has the INDARA keyword specified in the DDS, you must not specify PASS(*NOIND). If it does not, you must specify PASS(*NOIND).

PGMNAME(program_name)

The PGMNAME keyword identifies the program that is to handle the support for the special I/O device (indicated by a Device-Entry of SPECIAL).

Note: The parameter must be a valid program name and not a bound procedure name.

See "Positions 36-42 (Device)" on page 290 and "PLIST(Plist_name)" for more information.

PLIST(Plist_name)

The PLIST keyword identifies the name of the parameter list to be passed to the program for the SPECIAL file. The parameters identified by this entry are added to the end of the parameter list passed by the program. (The program is specified using the PGMNAME keyword, see "PGMNAME(program_name).") This keyword can only be specified when the Device-Entry (positions 36 to 42) in the file description line is SPECIAL.

PREFIX(prefix{:nbr_of_char_replaced})

The PREFIX keyword is used to partially rename the fields in an externally described file.The characters specified in the first parameter are prefixed to the names of all fields defined in all records of the file specified in positions 7-16. The characters can be specified as a name, for example PREFIX(F1_), or as a character literal, for example PREFIX('F1_'). A character literal must be used if the prefix contains a period, for example PREFIX('F1DS.') or PREFIX('F1DS.A'). To remove characters from the beginning of every name, specify an empty string as the first parameter: PREFIX('':number_to_remove). 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.

The 'nbr_of_char_replaced' parameter is not used when applying the prefix to an alias name. See the ALIAS keyword for information on how the PREFIX keyword interacts with the ALIAS keyword.

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 an 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 contain a period or end in a period. In this case, the field names must all be subfields of the same qualified data structure. The data structure must be defined as a qualified data structure. For example, for PREFIX('F1DS.'), data structure F1DS must be define as a qualified data structure; if the file has fields FLD1 and FLD2, the data structure must have subfields F1DS.FLD1 and F1DS.FLD2. Similarly, for PREFIX('F2DS.A'), data structure F2DS must be a qualified data structure; if the file has fields FLD1 and FLD2, the data structure must have subfields F2DS.AFLD1 and F2DS.AFLD2.
- If the prefix is a character literal, it must be uppercase.
- If an externally-described data structure is used to define the fields in the file, care must be taken to ensure that the field names in the file are the same as the subfield names in the data structure. The following table shows the prefix required for an externally-described file and externally-described data structure for several prefixed versions of the name "XYNAME". When the "Internal name" column contains a dot, for example D1.NAME, the externally-described data structure is defined as QUALIFIED, and the PREFIX for the File specification must contain a dot.

PREFIX for file	PREFIX for externally-described data structure	Internal name
PREFIX(A)	PREFIX(A)	AXYNAME
PREFIX(A:2)	PREFIX(A:2)	ANAME
PREFIX('D.')	None	D.XYNAME
PREFIX('D.' : 2)	PREFIX('' : 2)	D.NAME
PREFIX('D.A')	PREFIX(A)	D.AXYNAME
PREFIX('D.A' : 2)	PREFIX(A : 2)	D.ANAME
PREFIX('':2)	PREFIX('' : 2)	NAME

Examples:

The following example adds the prefix "NEW_" to the beginning of the field names for file NEWFILE, and the prefix "OLD_" to the beginning of the field names for file OLDFILE.

```
Fnewfile   o    e            disk    prefix(NEW_)
Foldfile   if   e            disk    prefix(OLD_)
C                   READ      OLDREC
C                   EVAL      NEWIDNO = OLD_IDNO
C                   EVAL      NEWABAL = OLD_ABAL
C                   WRITE     NEWREC
```

The following example uses PREFIX(N:2) on both file FILE1 and the externally-described data structure DS1. The File-specification prefix will cause the FILE1 fields XYIDNUM and XYCUSTNAME to be known as NIDNUM and NCUSTNAME in the program; the Data-specification prefix will cause the data structure to have subfields NIDNUM and NCUSTNAME. During the READ operation, data from the record will be moved to the subfields of DS1, which can then be passed to the subprocedure processRec to process the data in the record.

```
Ffile1     if   e             disk    prefix(N:2)
D ds1           e ds                  extname(file1) prefix(N:2)
C                   READ      file1
C                   CALLP     processRec (ds1)
```

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.')
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 MYDS2. 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)
D myds2         e ds                  qualified extname(myfile)
D                                     prefix('F2':3)
```

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. The default, *COMPAT not specified, will require the use of the extended length data structure.

Extended Length PRTCTL Data Structure

A minimum of 15 bytes is required for this data structure. 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 (valid entries: blank or 0-255)

4-6 A three-position character field that contains the space-after value (valid entries: 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 (valid entries: 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 (valid entries: 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 (valid entries: 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 (valid entries: 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 contained 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 ILE RPG compiler does not initialize subfield 5 until after the first output line is printed. The compiler then changes subfield 5 after each output operation to the file.

QUALIFIED

The QUALIFIED keyword controls how the record formats for the file are specified in your RPG source.

If this keyword is specified, the record formats must be qualified with the file name when they are specified in the RPG source; for example format FMT1 in qualified file FILE1 must be specified as FILE1.FMT1. The record format names can be the same as other names used within the RPG source.

If this keyword is not specified, the record formats must not be qualified with the file name; format FMT1 is specified as FMT1. The record format names must be unique names within the RPG source.

Rules for the QUALIFIED keyword:

- When a file is qualified, its record names must be qualified everywhere in the source except when specified as parameters of the File specification keywords RENAME, INCLUDE, IGNORE, and SFILE. The name must not be qualified when specified as the parameter of those keywords.
- When a file is qualified, Input and Output specifications are not allowed or generated for the file. This means that external fields from the file are not automatically defined as fields in the program. All I/O must be done with result data structures.
- The QUALIFIED keyword is valid only for externally-described files.
- The QUALIFIED keyword cannot be specified with the LIKEFILE keyword; files defined with LIKEFILE always have qualified record formats.

```
 * file1 has formats HDR, INFO, ERR.
 * file2 has format INFO.
 * The QUALIFIED keyword is used for both files, making it
 * unnecessary to rename one of the "INFO" formats.

 * Note that the record format names are not qualified when
 * specified in keywords of the File specification.
Ffile1     if   e             disk qualified
F                                  ignore(hdr)
F                                  rename(err:errorRec)
Ffile2     o    e             disk qualified
 * The record formats must be qualified on all specifications other
 * than the File specification for the file.
D ds1             ds                 likerec(file1.info : *input)
D errDs           ds                 likerec(file1.errorRec : *input)
D ds2             ds                 likerec(file2.info : *output)
 /free
       read file1.info ds1;
       eval-corr ds2 = ds1;
       write file2.info ds2;
       read file1.errorRec errDs;
```

Figure 116. Example of the QUALIFIED keyword

RAFDATA(filename)

The RAFDATA keyword identifies the name of the input or update file that contains the data records to be processed for a Record Address File (RAF) (an R in position 18). See “Record Address File (RAF)” on page 282 for further information.

RECNO(fieldname)

The RECNO keyword specifies that a DISK file is 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.

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.

The compiler will not open a SEQ or DISK file for blocking or unblocking records if the RECNO keyword is specified for the file. Note that the keywords RECNO and BLOCK(*YES) cannot be specified for the same file.

Note: 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.

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.

Remember that for a qualified file, the unqualified form of the record format name is used for both parameters of the RENAME keyword.

SAVEDS(DSname)

The SAVEDS keyword allows the specification of the data structure saved and restored for each device. Before an input operation, the data structure for the device operation is saved. After the input operation, the data structure for the device associated with this current input operation is restored. This data structure cannot be a data area data structure, file information data structure, or program status data structure, and it cannot contain a compile-time array or prerun-time array.

If the SAVEDS keyword is not specified, no saving and restoring is done. SAVEDS must not be specified for shared files.

When you specify SAVEDS but not MAXDEV, the ILE RPG program assumes a multiple device file (MAXDEV with a parameter of *FILE).

SAVEIND(number)

The SAVEIND keyword specifies the number of indicators that are to be saved and restored for each device attached to a mixed or multiple device file. Before an input operation, the indicators for the device associated with the previous input or output operation are saved. After the input operation, the indicators for the device associated with this current input operation are restored.

Specify a number from 1 through 99, as the parameter to the SAVEIND keyword. No indicators are saved and restored if the SAVEIND keyword is not specified, or if the MAXDEV keyword is not specified or specified with the parameter *ONLY.

If you specified the DDS keyword INDARA, the number you specify for the SAVEIND keyword must be less than any response indicator you use in your DDS. For example, if you specify INDARA and CF01(55) in your DDS, the maximum value for the SAVEIND keyword is 54. The SAVEIND keyword must not be used with shared files.

The INDDS keyword cannot be specified with this keyword.

When you specify the SAVEIND keyword but not the MAXDEV keyword, the ILE RPG program assumes a multiple device file.

SFILE(recformat:rrnfield)

The SFILE keyword is used to define internally the subfiles that are specified in an externally described WORKSTN file. The recformat parameter identifies the RPG IV name of the record format to be processed as a subfile. The rrnfield parameter identifies the name of the relative-record number field for this subfile. You must specify an SFILE keyword for each subfile in the DDS.

If you define a display file like another file using the LIKEFILE keyword, and the parent file has subfiles, then you must specify the SFILE keyword for each subfile in the new file, so that you can provide the names of the relative record number fields for the subfiles.

If a file is defined with the TEMPLATE keyword, the rrnfield parameter of the SFILE keyword is not specified.

The relative-record number of any record retrieved by a READC or CHAIN operation is placed into the field identified by the rrnfield parameter. This field is also used to specify the record number that RPG IV uses for a WRITE operation to the subfile or for output operations that use ADD. The field name specified as the rrnfield parameter must be defined as numeric with zero decimal positions. The field must have enough positions to contain the largest record number for the file. (See the SFLSIZ keyword in the iSeries Information Center database and file systems category.)

Relative record number processing is implicitly defined as part of the SFILE definition. If multiple subfiles are defined, each subfile requires the specification of the SFILE keyword.

Do not use the SFILE keyword with the SLN keyword.

Remember that for a qualified file, the unqualified form of the record format name is used for the first parameter of the SFILE keyword.

SLN(number)

The SLN (Start Line Number) keyword determines where a record format is written to a display file. The main file description line must contain WORKSTN in positions 36 through 42 and a C or O in positions 17. The DDS for the file must specify the keyword SLNO(*VAR) for one or more record formats. When you specify the SLN keyword, the parameter will automatically be defined in the program as a numeric field with length of 2 and with 0 decimal positions.

Do not use the SLN keyword with the SFILE keyword.

STATIC

The STATIC keyword indicates that the RPG file control information is kept in static storage; all calls to the subprocedure use the same RPG file control information. The RPG file control information holds its state across calls to the subprocedure. If the file is open when the subprocedure ends, then the file will still be open on the next call to the subprocedure.

When the STATIC keyword is not specified, the RPG file control information is kept in automatic storage; each call to the subprocedure uses its own version of the RPG file control information. The RPG file control information is initialized on every call to the subprocedure. If the file is open when the subprocedure ends, then the file will be closed when the subprocedure ends.

Rules for the STATIC keyword:

- The STATIC keyword can only be specified for file definitions in subprocedures. The STATIC keyword is implied for files defined in global definitions.
- A file defined with the STATIC keyword will remain open until it is explicitly closed by a CLOSE operation, or until the activation group ends.
- If a File Information Data Structure (INFDS) is defined for the file, the specification of the STATIC keyword for the data structure must match the specification of the STATIC keyword for the file.

```
P numInStock      b                   export
 * File "partInfo" is defined as STATIC.  The file will be
 * opened the first time the procedure is called, because
 * the USROPN keyword is not specified.
 * Since there is no CLOSE operation for the file, it
 * will remain open until the activation group ends.
FpartInfo  if   e           k disk    static
 * File "partErrs" is not defined as STATIC, and the USROPN
 * keyword is used.  The file will be opened by the OPEN
 * operation, and it will be closed automatically when the
 * procedure ends.
FpartErrs  o    e             disk    usropn

D numInStock      pi            10i 0
D    id_no                      10i 0 value
D partInfoDs      ds                  likerec(partRec:*input)
D partErrDs       ds                  likerec(errRec:*output)

 /free
       // Search for the input value in the file
       chain id_no partRrec partInfoDs;
       if  not %found(partInfo);
          // write a record to the partErrs file indicating
          // that the id_no record was not found.  The
          // file must be opened before the record can
          // be written, since the USROPN keyword was
          // specified.
          partErrDs.id_no = id_no;
          open partErrs;
          write errRec partErrDs;
          return -1; // unknown id
       endif;
       return partInfoDs.qty;
 /end-free
P numInStock      e
```

Figure 117. Example of the STATIC keyword for a File specification

TEMPLATE

The TEMPLATE keyword indicates that this file definition is to be used only at compile time. Files defined with the TEMPLATE keyword are not included in the program. The template file can only be used as a basis for defining other files later in the program using the LIKEFILE keyword.

Rules for the TEMPLATE keyword:

- The RPG symbol name for the template file can be used only as the parameter of a LIKEFILE keyword on a file specification, or a LIKEFILE keyword on a Definition specification.
- The RPG symbol name of a record format of a template file can be used only as the parameter of a LIKEREC Definition keyword.
- Keywords that are not inherited by LIKEFILE definitions are not allowed for a template file.

  See Table 49 on page 301 for more information.

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 file. If the file on which this keyword is specified is indexed and the key field is a time, then the time format specified also provides the default external format for the key field.

For a Record-Address file this specifies the external time format of time limits keys read from the record-address file.

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 36 on page 209 for valid format and separators. For more information on external formats, see "Internal and External Formats" on page 179.

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 primary, secondary, table, or record-address files, or for output files conditioned by the 1P (first page) indicator.

The USROPN keyword is required for programmer control of only the first file opening. For example, if a file is opened and later closed by the CLOSE operation, the programmer can reopen the file (using the OPEN operation) without having specified the USROPN keyword on the file description specification.

See also "EXTIND(*INUx)" on page 296.

File Types and Processing Methods

Table 50 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
- Sequential-within-limits processing.

Table 50. 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 cycle	Blank	Blank	I	Access by key sequentially
Sequential	Within Limits	READ READE READP READPE cycle	L	A, P, G, D, T, Z, or F	I	Access by key sequentially controlled by record-address-limits file

Table 50. Processing Methods for DISK Files (continued)

Access	Method	Opcode	Position 28	Position 34	Position 35	Explanation
Sequential	RRN	READ cycle	Blank	Blank	T	Access sequentially restricted to RRN numbers in record-address file

For further information on the various file processing methods, see the section entitled "Methods for Processing Disk Files", in the chapter "Accessing Database Files" in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Chapter 14. Definition Specifications

Definition specifications can be used to define:

- Standalone fields
- Named constants
- Data structures and their subfields
- Prototypes
- Procedure interface
- Prototyped parameters

For more information on data structures, constants, prototypes, and procedure interfaces, see also Chapter 7, "Defining Data and Prototypes," on page 125 For more information on data types and data formats, see also Chapter 9, "Data Types and Data Formats," on page 179.

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 also Chapter 8, "Using Arrays and Tables," on page 159.

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 cycle-main procedure). For more information on scope, see "Scope of Definitions" on page 24.

A built-in function (BIF) can be used in the keyword field as a parameter to a keyword. It is allowed on the definition specification only if the values of all arguments are known at compile time. When specified as parameters for the definition specification keywords DIM, OCCURS, OVERLAY, and PERRCD, all arguments for a BIF must be defined earlier in the program. For further information on using built-in functions, see "Built-in Functions" on page 430.

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 118. 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 119. 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 120. Definition Specification Continued Name Line Layout

Position 6 (Form Type)

Enter a D in this position for definition specifications.

Positions 7-21 (Name)

Entry **Explanation**

Name The name of the item being defined.

Blank Specifies filler fields in data-structure subfield definitions, or an unnamed data structure in data-structure definitions.

The normal rules for RPG IV symbolic names apply; reserved words cannot be used (see “Symbolic Names” on page 3). The name can begin in any position in the space provided. Thus, 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.)

TIP

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 ILE C.

Position 22 (External Description)

This position is used to identify a data structure or data-structure subfield as externally described. If a data structure or subfield is not being 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 used.
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.

RPG IV retrieves the data area at initialization and rewrites it at end of 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.
- If a name is not specified either by the DTAARA keyword, or by positions 7-21, *LDA (the local data area) is used as the name of the external data area.

Positions 24-25 (Definition Type)

Entry **Explanation**

Blank The specification defines either a data structure subfield or a parameter within a prototype or procedure interface definition.

C The specification defines a constant. Position 25 must be blank.

DS The specification defines a data structure.

PR The specification defines a prototype and the return value, if any.

PI The specification defines a procedure interface, and the return value if any.

S The specification defines a standalone field, array or table. Position 25 must be blank.

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.

For a list of valid keywords, grouped according to type of definition, please refer to Table 52 on page 371.

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 9999999, 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, parameter, or subfield is being defined LIKE another field, or
- the standalone field, parameter, 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 9999999, 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.

Notes:

1. 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: 1-63; for binary numeric fields: 1-9; for integer and unsigned numeric fields: 3, 5, 10, or 20;).
2. For float numeric fields the number specified is the number of bytes, NOT the number of digits (4 or 8 bytes).

3. If you want to define a character, UCS-2 or graphic definition with a length greater than 9999999, use the LEN keyword instead of specifying the Length entry. If you want to explicitly position a subfield whose length is defined with the LEN keyword, use the OVERLAY keyword. You code the data structure name in the first parameter of the OVERLAY keyword, and the desired start position of the subfield in the second parameter of the OVERLAY keyword.

+|-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.

Notes:

1. 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.
2. For float fields, the entry must be blank or +0. The size of a float field cannot be changed as with other numerics.

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 318, 'Reserved Words'.

Position 40 (Internal Data Type)

This entry allows you to specify how a standalone field, parameter, 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 When the LIKE keyword is not specified:

- If the decimal positions entry is blank, then the item is defined as character
- If the decimal positions entry is not blank, then the item is defined as packed numeric if it is a standalone field or parameter; 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)

B Numeric (Binary format)

C UCS-2 (Fixed or Variable-length format)

D Date

F Numeric (Float format)

G Graphic (Fixed or Variable-length format)

I Numeric (Integer format)

N Character (Indicator format)

O Object

P Numeric (Packed decimal format)

S Numeric (Zoned format)

T Time

U Numeric (Unsigned format)

Z Timestamp

***** 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 non-float numeric, there must always be an entry in these positions. If there are no decimal positions enter a zero (0) in position 42. For example, an integer or unsigned field (type I or U in position 40) requires a zero for this entry.

Entry	Explanation
Blank	The value is not numeric (unless it is a float field) or has been defined with the LIKE keyword.
0-63	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 data base 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. Use this area to specify any keywords necessary to fully define the field.

Definition-Specification Keywords

Definition-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 definition-specification keywords, the keyword field can be continued on subsequent lines. See “Definition Specification Keyword Continuation Line” on page 316 and “Definition Specification Keyword Field” on page 252.

ALIAS

When the ALIAS keyword is specified for an externally-described data structure, the RPG compiler will use the alias (alternate) names for the subfields, if present. If the ALIAS keyword is not specified for the data structure, or an external field does not have an alias name defined, the RPG compiler will use the standard external field name.

When alias names are being used and you want to rename a subfield, you specify the alias name as the parameter to the EXTFLD keyword. The EXTFLD keyword does not support continuation, so you must specify the entire name on one source specification. Figure 121 on page 323 shows an example with two data structures, defined for the same file. The data structure that has the ALIAS keyword coded uses the alias name, CUSTOMER_ADDRESS, as the parameter for the EXTFLD keyword. The data structure that does not have the ALIAS keyword coded uses the standard name, CUSTAD, as the parameter for the EXTFLD keyword.

Note: If the alternate name for a particular external field is enclosed in quotes, the standard external field name is used for that field.

When the PREFIX keyword is specified with the ALIAS keyword, the second parameter of PREFIX, indicating the number of characters to be replaced, does not apply to the alias names. In the following discussion, assume that the external file MYFILE has fields XYCUSTNM and XYID_NUM, and the XYCUSTNM field has the alias name CUSTOMER_NAME.

- If keyword PREFIX(NEW_) is specified, there is no second parameter, so no characters will be replaced for any names. The names used for the RPG subfields will be NEW_CUSTOMER_NAME and NEW_XYID_NUM.
- If keyword PREFIX(NEW_:2) is specified, two characters will be removed from the names of fields that do not have an alias name. The names used for the RPG subfields will be NEW_CUSTOMER_NAME and NEW_ID_NUM. The first two characters, "XY", are replaced in XYID_NUM, but no characters are replaced in CUSTOMER_NAME.

- If keyword PREFIX('':2) is specified, two characters will be removed the names of fields that do not have an alias name. The names used for the RPG subfields will be CUSTOMER_NAME and ID_NUM. The first two characters, "XY", are replaced in XYID_NUM, but no characters are replaced in CUSTOMER_NAME.

```
 * The DDS specifications for file MYFILE, using the ALIAS keyword
 * for the first two fields, to associate alias name CUSTOMER_NAME
 * with the CUSTNM field and alias name CUSTOMER_ADDRESS
 * with the CUSTAD field.
A          R CUSTREC
A            CUSTNM        25A         ALIAS(CUSTOMER_NAME)
A            CUSTAD        25A         ALIAS(CUSTOMER_ADDRESS)
A            ID_NUM        12P 0

 * The RPG source, using the ALIAS keyword.
 * The customer-address field is renamed to CUST_ADDR
 * for both data structures.
D aliasDs       e ds                  ALIAS
D                                     QUALIFIED EXTNAME(myfile)
D   cust_addr   e                     EXTFLD(CUSTOMER_ADDRESS)
D noAliasDs     e ds
D                                     QUALIFIED EXTNAME(myfile)
D   cust_addr   e                     EXTFLD(CUSTAD)
 /free
   // The ALIAS keyword is specified for data structure "aliasDs"
   // so the subfield corresponding to the "CUSTNM" field has
   // the alias name "CUSTOMER_NAME"
   aliasDs.customer_name = 'John Smith';
   aliasDs.cust_addr = '123 Mockingbird Lane';
   aliasDs.id_num = 12345;

   // The ALIAS keyword is not specified for data structure
   // "noAliasDs", so the subfield corresponding to the "CUSTNM"
   // field does not use the alias name
   noAliasDs.custnm = 'John Smith';
   aliasDs.cust_addr = '123 Mockingbird Lane';
   noAliasDs.id_num = 12345;
```

Figure 121. Using the ALIAS keyword for an externally-described data structure

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 16-byte boundary whether or not ALIGN is specified.

See “Aligning Data Structure Subfields” on page 140 for more information.

ALT(array_name)

The ALT keyword is used to indicate that the compile-time or 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.

ALTSEQ(*NONE)

When the ALTSEQ(*NONE) keyword is specified, the alternate collating sequence will not be used for comparisons involving this field, even when the ALTSEQ keyword is specified on the control specification. ALTSEQ(*NONE) on Data Definition specifications will be meaningful only if one of ALTSEQ, ALTSEQ(*SRC) or ALTSEQ(*EXT) is coded in the control specifications. It is ignored if this is not true.

ALTSEQ(*NONE) is a valid keyword for:
- Character standalone fields
- Character arrays
- Character tables
- Character subfields
- Data structures
- Character return values on Procedure Interface or Prototype definitions
- Character Prototyped Parameters

ASCEND

The ASCEND keyword is used to describe the sequence of the data in any of the following:
- An array
- A table loaded at prerun-time or compile time
- A prototyped parameter

See also “DESCEND” on page 327.

Ascending sequence means that the array or table entries must start with the lowest data entry (according to the collating sequence) and go to the highest. Items with equal value are allowed.

A prerun-time 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 RPG IV exception/error handling routine. A run-time array (loaded by input and/or calculation specifications) is not sequence checked.

When ALTSEQ(*EXT) is specified, the alternate collating sequence is used when checking the sequence of compile-time arrays or tables. If the alternate sequence is not known until run-time, the sequence is checked at run-time; if the array or table is out of sequence, control passes to the RPG IV exception/error handling routine.

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.

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. A valid graphic CCSID is 65535 or a CCSID with the EBCDIC double-byte encoding scheme (X'1200'). A valid UCS-2 CCSID has the UCS-2 encoding scheme (x'7200').

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.

class-name must be a constant character value.

CONST{(constant)}

The CONST keyword is used

- To specify the value of a named constant
- To indicate that a parameter passed by reference is read-only.

When specifying the value of a named constant, the CONST keyword itself is optional. That is, the constant value can be specified with or without the CONST keyword.

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 249 for further details).

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, and optionally the separator character, for any of these items of type Date: standalone field;

data-structure subfield; prototyped parameter; or return value on a prototype or procedure-interface definition. This keyword will be automatically generated for an externally described data structure subfield of type Date and determined at compile time.

If DATFMT is not specified, the Date field will have the date format and separator as specified by the DATFMT keyword on the control specification, if present. If none is specified on the control specification, then it will have *ISO format.

See Table 33 on page 207 for valid formats and separators. For more information on internal formats, see "Internal and External Formats" on page 179.

DESCEND

The DESCEND keyword describes the sequence of the data in any of the following:
- An array
- A table loaded at prerun-time or compile time
- A prototyped parameter

See also "ASCEND" on page 324.

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 prerun-time 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 RPG IV exception/error handling routine. A run-time array (loaded by input and/or calculation specifications) is not sequence checked.

When ALTSEQ(*EXT) is specified, the alternate collating sequence is used when checking the sequence of compile-time arrays or tables. If the alternate sequence is not known until run-time, the sequence is checked at run-time; if the array or table is out of sequence, control passes to the RPG IV exception/error handling routine.

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.

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.

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 "*DTAARA DEFINE" on page 653).

The DTAARA keyword can only be used in the main source section. It cannot be used in a subprocedure.

You can create three kinds of data areas:
- *CHAR Character
- *DEC Numeric
- *LGL Logical

You can also create a DDM data area (type *DDM) that points to a data area on a remote system of one of the three types above.

Only character and numeric types (excluding float numeric) are allowed to be associated with data areas. The actual data area on the system must be of the same type as the field in the program, with the same length and decimal positions. Indicator fields can be associated with either a logical data area or a character data area. If you want to store other types in a data area, you can use a data structure for the data area, and code the subfields of any type, except pointers. Pointers cannot be stored in data areas.

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 neither the parameter nor the data-structure name is specified, then the default is *LDA.

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, passing the variable as an entry parameter, or sharing the variable with another program through the IMPORT and EXPORT keywords.

When the DTAARA keyword is specified, the IN, OUT, and UNLOCK operation codes can be used on the data area.

EXPORT{(external_name)}

The specification of the EXPORT keyword allows a globally defined data structure or standalone field defined within a module to be used by another module in the program. The storage for the data item is allocated in the module containing the EXPORT definition. The external_name parameter, if specified, must be a character literal or constant.

The EXPORT keyword on the definition specification is used to export data items and cannot be used to export procedure names. To export a procedure name, use the EXPORT keyword on the procedure specification.

Note: The initialization for the storage occurs when the program entry procedure (of the program containing the module) is first called. RPG IV will not do any further initialization on this storage, even if the procedure ended with LR on, or ended abnormally on the previous call.

The following restrictions apply when EXPORT is specified:

- Only one module may define the data item as exported
- You cannot export a field that is specified in the Result-Field entry of a PARM in the *ENTRY PLIST
- Unnamed data structures cannot be exported
- BASED data items cannot be exported
- The same external field name cannot be specified more than once per module and also cannot be used as an external procedure name
- IMPORT and EXPORT cannot both be specified for the same data item.

For a multiple-occurrence data structure or table, each module will contain its own copy of the occurrence number or table index. An OCCUR or LOOKUP operation in any module will have only a local impact since the occurrence number or index is local to each module.

See also "IMPORT{(external_name)}" on page 337.

TIP

The keywords IMPORT and EXPORT allow you to define a "hidden" interface between modules. As a result, use of these keywords should be limited only to those data items which are global throughout the application. It is also suggested that this global data be limited to things like global attributes which are set once and never modified elsewhere.

EXTFLD(field_name)

The EXTFLD keyword is used to rename a subfield in an externally described data structure. Enter the external name of the subfield as the parameter to the EXTFLD keyword, and specify the name to be used in the program in the Name field (positions 7-21).

The external name can be either a simple name or a character literals. If a character literal is specified, the external name name name must be specified in the correct case. For example, if the external name is MYFIELD, the file-name parameter could be specified as a name in mixed case such as myField or myfield, but if specified as a literal it must be 'MYFIELD'.

If the name is not a valid simple RPG name, it must be specified as a literal. For example, to rename external field A.B, specify EXTFLD('A.B').

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. Figure 121 on page 323 shows an example of the EXTFLD keyword with the ALIAS keyword.

EXTFMT(code)

The EXTFMT keyword is used to specify the external data type for compile-time and prerun-time 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.

Note: The values specified for EXTFMT will apply to the files identified in both the TOFILE and FROMFILE keywords, even if the specified names are different.

The possible values for the parameter are:

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.

P The data for the array or table is in packed decimal format.

S The data for the array or table is in zoned decimal format.

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. 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.
4. 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. If graphic data is included in the data, the presence of double-byte data and the shift-out and shift-in characters in the data will reduce the actual amount of data that can be placed in the array element; the rest of the element will be padded with blanks. For example, for a 4-character UCS-2 array, only one double-byte character can be specified in the compile-time data; if the compile-time data were 'oXXi', where 'XX' is converted to the UCS-2 character U'yyyy', the UCS-2 element would contain the value U'yyyy002000200020'.

EXTNAME(file-name{:format-name}{:*ALL|*INPUT|*OUTPUT|*KEY})

The EXTNAME keyword is used to specify 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 file-name and format-name parameters can be either names or character literals. If a character literal is specified, the file or format name name must be specified in the correct case. For example, if the external file is MYFILE, the file-name parameter could be specified as a name in mixed case such as myFile or myfile, but if specified as a literal it must be 'MYFILE'. If the file-name is a character literal, it can be in any of the following forms

```
'LIBRARY/FILE'
'FILE'
'*LIBL/FILE'
```

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 is 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 position 22, and the EXTNAME keyword is not specified, the name specified in positions 7-21 is used.

The compiler will generate the following definition specification entries for all fields of the externally described data structure:

- Subfield name (Name will be the same as the external name, unless the ALIAS keyword is specified for the data structure, or the is field renamed by the EXTFLD keyword, or the PREFIX keyword on a definition specification is used to apply a prefix).
- Subfield length
- Subfield internal data type (will be the same as the external type, unless the CVTOPT control specification keyword or command parameter 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 program whose prototype is being defined. The name can be a character constant or a character variable. When EXTPGM is specified, then a dynamic call will be done.

If neither EXTPGM or EXTPROC is specified for a prototype, 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 or procedure interface with EXTPGM must be passed by reference. In addition, you cannot define a return value.

EXTPROC({*CL|*CWIDEN|*CNOWIDEN| {*JAVA:class-name:}}name)

The EXTPROC keyword can have one of the following formats:

EXTPROC(*CL:name)
Specifies an external procedure that is written in ILE CL, or an RPG procedure to be called by ILE CL. Use *CL if your program uses return values with data types that CL handles differently from RPG. For example, use *CL when prototyping an RPG procedure that is to be called by a CL procedure when the return value is 1A.

EXTPROC(*CWIDEN:name | *CNOWIDEN:name)
Specifies an external procedure that is written in ILE C, or an RPG procedure to be called by ILE C.

Use *CNOWIDEN or *CWIDEN if your program uses return values or parameters passed by value with data types that C handles differently from RPG. Use *CWIDEN or *CNOWIDEN when defining an RPG procedure that is to be called by C, or when defining the prototype for a C procedure, where the returned value or a parameter passed by value is 1A, 1G or 1C, 5U, 5I, or 4F.

Use *CNOWIDEN if the ILE C source contains `#pragma argument(procedure-name,nowiden)` for the procedure; otherwise, use *CWIDEN.

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 *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

EXTPROC(name)
Specifies an external procedure that is written in or to be called by RPG or COBOL. This format should also be used for a procedure that can be called by any of RPG, COBOL, C, or CL; in this case, you must ensure that the return value and the parameters do not have any of the problems listed above for *CL, *CWIDEN, and *CNOWIDEN.

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, a bound call will be done.

If neither EXTPGM or EXTPROC is 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 or procedure name, if neither EXTPGM or EXTPROC is specified) starts with "CEE" or an underscore ('_'), the compiler will treat this as a system built-in. To avoid confusion with system provided APIs, you should not name your procedures starting with "CEE".

For example, to define the prototype for the procedure SQLAllocEnv, that is in the service program QSQCLI, the following definition specification could be coded:

```
D SQLEnv          PR                  EXTPROC('SQLAllocEnv')
```

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.

When a prototype is specified for a procedure, the EXTPROC keyword is specified for the prototype. Otherwise, the EXTPROC keyword is specified for the procedure interface. It is only necessary to explicitly specify a prototype when the procedure will be called from another RPG module. When the procedure is only called from within the same module, or when it is only called by non-RPG callers, the prototype can be implicitly derived from the procedure interface.

Figure 122 on page 334 shows an example of the EXTPROC keyword with a procedure pointer as its parameter.

```
 * Assume you are calling a procedure that has a procedure
 * pointer as the EXTPROC. Here is how the prototype would
 * be defined:
D DspMsg          PR            10A   EXTPROC(DspMsgPPtr)
D  Msg                       32767A
D  Length                        4B 0 VALUE
 * Here is how you would define the prototype for a procedure
 * that DspMsgPPtr could be assigned to.
D MyDspMsg        PR                  LIKE(DspMsg)
D  Msg                       32767A
D  Length                        4B 0 VALUE
 * Before calling DSPMSG, you would assign DSPMSGPPTR
 * to the actual procedure name of MyDspMsg, that is
 * MYDSPMSG.
C                     EVAL      DspMsgPPtr = %paddr('MYDSPMSG')
C                     EVAL      Reply = DspMsg(Msg, %size(Msg))
 ...
P MyDspMsg       B
```

Figure 122. Using EXTPROC with a Procedure Pointer

```
    char RPG_PROC (short s, float f);
    char C_PROC (short s, float f);
    #pragma argument(RPG_PROC, nowiden)
    #pragma argument(C_PROC, nowiden)

    /* "fn" calls the RPG procedure with unwidened parameters,       */
    /* and expects the return value to be passed according to C      */
    /* conventions.                                                  */
    void fn(void)
    {
       char c;

       c = RPG_PROC(5, 15.3);
    }

    /* Function C_PROC expects its parameters to be passed unwidened.*/
    /* It will return its return value using C conventions.          */
    char C_PROC (short s, float f);
    {
       char c = 'x';

       if (s == 5 || f < 0)
       {
          return 'S';
       }
       else
       {
          return 'F';
       }
    }
```

*Figure 123. Using EXTPROC with *CNOWIDEN - C Code*

```
D RPG_PROC        PR             1A   EXTPROC(*CNOWIDEN : 'RPG_PROC')
D   short                        5I 0 VALUE
D   float                        4F   VALUE

D C_RPOC          PR             1A   EXTPROC(*CNOWIDEN : 'C_PROC')
D   short                        5I 0 VALUE
D   float                        4F   VALUE

P RPG_PROC        B                   EXPORT
D                 PI             1A
D   short                        5I 0 VALUE
D   float                        4F   VALUE

D   char          S              1A

 * Call the C procedure
C                   EVAL      c = C_PROC(4 : 14.7)

 * Return the value depending on the values of the parameters
C                   IF        short < float
C                   RETURN    'L'
C                   ELSE
C                   RETURN    'G'
C                   ENDIF

P                 E
```

*Figure 124. Using EXTPROC with *CNOWIDEN - RPG Code*

```
   char RPG_PROC (short s, float f);
   char C_PROC (short s, float f);

   /* Function "fn" calls the RPG procedure with widened parameters,*/
   /* and expects the return value to be passed according to C      */
   /* conventions.                                                  */
   void fn(void)
   {
      char c;

      c = RPG_PROC(5, 15.3);
   }

   /* Function C_PROC expects its parameters to be passed widened.  */
   /* It will return its return value using C conventions.          */
   char C_PROC (short s, float f);
   {
      char c = 'x';

      if (s == 5 || f < 0)
      {
         return 'S';
      }
      else
      {
         return 'F';
      }
   }
```

*Figure 125. Using EXTPROC with *CWIDEN - C Code*

```
D RPG_PROC        PR             1A   EXTPROC(*CWIDEN : 'RPG_PROC')
D   short                        5I 0 VALUE
D   float                        4F   VALUE

D C_PROC          PR             1A   EXTPROC(*CWIDEN : 'C_PROC')
D   short                        5I 0 VALUE
D   float                        4F   VALUE

P RPG_PROC        B                   EXPORT
D                 PI             1A
D   short                        5I 0 VALUE
D   float                        4F   VALUE

D   char          S              1A

 * Call the C procedure
C                   EVAL      c = C_PROC(4 : 14.7)

 * Return the value depending on the values of the parameters
C                   IF        short < float
C                   RETURN    'L'
C                   ELSE
C                   RETURN    'G'
C                   ENDIF

P                 E
```

*Figure 126. Using EXTPROC with *CWIDEN - RPG Code*

```
         /* CL procedure CL_PROC */
         DCL &CHAR1 TYPE(*CHAR) LEN(1)

         /* Call the RPG procedure */
         CALLPRC RPG_PROC RTNVAR(&CHAR1)
```

*Figure 127. Using EXTPROC with *CL - CL Code*

```
D RPG_PROC        PR             1A   EXTPROC(*CL : 'RPG_PROC')

P RPG_PROC        B                   EXPORT
D                 PI             1A

C                   RETURN    'X'

P                 E
```

*Figure 128. Using EXTPROC with *CL - RPG Code*

```
P isValidCust     B                EXPORT
D                 PI            N    EXTPROC(*CL : 'isValidCust')
D   custId                    10A    CONST
D isValid         S             N    INZ(*OFF)
 /free
    ... calculations using the "custId" parameter
    return isValid;
 /end-free
P                 E
```

Figure 129. Using EXTPROC on a procedure interface for a procedure intended to be called only by CL callers

FROMFILE(file_name)

The FROMFILE keyword is used to specify the file with input data for the prerun-time array or table being defined. The FROMFILE keyword must be specified for every prerun-time array or table used in the program.

See also "TOFILE(file_name)" on page 369.

IMPORT{(external_name)}

The IMPORT keyword specifies that storage for the data item being defined is allocated in another module, but may be accessed in this module. The external_name parameter, if specified, must be a character literal or constant.

If a name is defined as imported but no module in the program contains an exported definition of the name, an error will occur at link time. See "EXPORT{(external_name)}" on page 329.

The IMPORT keyword on the definition specification is used to import data items and cannot be used to import procedure names. Procedure names are imported implicitly, to all modules in the program, when the EXPORT keyword is specified on a procedure specification.

The following restrictions apply when IMPORT is specified:

- The data item may not be initialized (the INZ keyword is not allowed). The exporting module manages all initialization for the data.
- An imported field cannot be defined as a compile-time or prerun-time array or table, or as a data area. (Keywords CTDATA, FROMFILE, TOFILE, EXTFMT, PERRCD, and DTAARA are not allowed.)
- An imported field may not be specified as an argument to the RESET operation code since the initial value is defined in the exporting module.
- You cannot specify an imported field in the Result-Field entry of a PARM in the *ENTRY PLIST.
- You cannot define an imported field as based (the keyword BASED is not allowed).
- This keyword is not allowed for unnamed data structures.
- The only other keywords allowed are DIM, EXTNAME, LIKE, OCCURS, and PREFIX.
- The same external field name cannot be specified more than once per module and also cannot be used as an external procedure name.

For a multiple-occurrence data structure or table, each module will contain its own copy of the occurrence number or table index. An OCCUR or LOOKUP operation in any module will have only a local impact since the occurrence number or index is local to each module.

INZ{(initial value)}

The INZ keyword initializes the standalone field, data structure, data-structure subfield, or object 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 *SYS, *JOB, *EXTDFT, *USER, *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 UCS-2 field may be initialized with a character, UCS-2 or graphic constant. If the constant is not UCS-2, the compiler will implicitly convert it to UCS-2 at compile time.

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.

Specifying INZ(*USER) intializes any character field or subfield to the name of the current user profile. Character fields must be at least 10 characters long. If the field is longer than 10 characters, the user name is left-justified in the field with blanks in the remainder.

Date fields can be initialized to *SYS or *JOB. Time and Timestamp fields can be initialized to *SYS.

Please see "Initialization of Nested Data Structures" on page 141 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 RPG default initial values (for example, blanks for character, 0 for numeric).
- Subfields of uninitialized data structures (INZ not specified on the definition specification for the data structure) are initialized to blanks (regardless of their data type).

This keyword is not valid in combination with BASED or IMPORT.

LEN(length)

The LEN keyword is used to define the length in characters of a Data Structure or character, UCS-2 or graphic definition. It is valid for Data Structure definitions, and for Prototype, Prototyped Parameter, Standalone Field and Subfield definitions where the type entry is A (Alphanumeric), C (UCS-2), or G (Graphic).

Rules for the LEN keyword:

- The data type A, C or G must be specified in the Data-Type entry.
- The LEN keyword cannot be specified if the Length entry is specified, or if the From and To entries are specified for subfields. The LEN keyword must be used to specify a length greater than 9,999,999.
- Length adjustment for LIKE definitions cannot be done using the LEN keyword.
- The length is specified in characters; for UCS-2 and Graphic definitions, each character represents two bytes.

```
 * Use the LEN keyword to define a standalone field of one million
 * characters and a standalone array of 100 characters.
D paragraph       S               A   LEN(1000000) VARYING(4)
D splitPara       S               A   LEN(100) DIM(10000)

 * Use the LEN keyword to define a data structure of length 16000000,
 * and to define three subfields.  Since the lengths of the parameters
 * are less than 9999999, they can be defined using from-and-to, or length
 * notation, or the LEN keyword.
D info            DS                  LEN(16000000)
D   name                          G   LEN(100) OVERLAY(info : 14000001)
D   address                   5000G   OVERLAY(info : 14000301)
D   country              1     40G

 * Use the LEN keyword to define a prototype that returns a varying
 * UCS-2 value that is up to 5000 UCS-2 characters long, and to define
 * two alphanumeric parameters.  Since the lengths of the parameters
 * are less than 9999999, they can be defined either using length notation
 * or the LEN keyword.
D getDftDir       PR              C   VARYING LEN(5000)
D   usrprf                        A   LEN(10) CONST
D   type                        5A   CONST
```

Figure 130. Examples of the LEN keyword

LIKE(name)

The LIKE keyword is used to define an item like an existing one. For information about using LIKE with an object, see "LIKE(object-name)" on page 341.

When the LIKE keyword is specified, the item being defined takes on the length and the 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 "*LIKE DEFINE" on page 651). 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 ALTSEQ(*NONE), 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.

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 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 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 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+++++++++++++++++++++++++++++
 *
 *  Define a field like another with a length increase of 5 characters.
 *
D  Name           S             20
D  Long_name      S             +5    LIKE(Name)
 *
 *  Define a data structure subfield array with DIM(20) like another
 *  field and initialize each array element with the value *ALL'X'.
 *  Also, declare another subfield of type pointer immediately
 *  following the first subfield. Pointer is implicitly defined
 *  with a length of 16 bytes
 *
D  Struct         DS
D   Dim20                             LIKE(Name) DIM(20) INZ(*ALL'X')
D   Pointer                       *
 *
 *  Define a field which is based on the *LDA.  Take the length and type
 *  of the field from the field 'Name'.
 *
D  Lda_fld        S                    LIKE(Name) DTAARA(*LDA)
```

Figure 131. 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 132. 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 parent data structure specified as the parameter to the LIKEDS keyword.

A data structure defined using LIKEDS is automatically qualified even if the parent data structure is not qualified. The subfields must be referred to using the qualified notation DSNAME.SUBFIELDNAME. If the parent data structure has any unnamed subfields, the child data structure will have the same unnamed subfields.

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 values of the ALIGN and ALTSEQ keywords 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).

```
 * Data structure qualDs is a qualified data structure
 * with two named subfields and one unnamed subfield
D qualDs          DS                  QUALIFIED
D  a1                           10A
D                                2A
D  a2                            5P 0 DIM(3)
 * Data structure unqualDs is a non-qualified data structure
 * with one named subfield and one unnamed subfield
D unqualDs        DS
D  b1                            5A
D                                5A
 * Data structure likeQual is defined LIKEDS(qualDs)
D likeQual        DS                  LIKEDS(qualDs)
 * Data structure likeUnqual is defined LIKEDS(unqualDs)
D likeUnqual      DS                  LIKEDS(unqualDs)
 /FREE
        // Set values in the subfields of the
        // parent data structures.
        qualDs.a1 = 'abc';
        qualDs.a2(1) = 25;
        b1 = 'xyz';

        // Set values in the subfields of the
        // child data structures.
        likeQual.a1 = 'def';
        likeQual.a2(2) = -250;
        likeUnqual.b1 = 'rst';

        // Display some of the subfields
        dsply likeQual.a1;  // displays 'def'
        dsply b1;           // displays 'xyz'
```

Figure 133. Defining data structures using 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 134. 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 135. Using a data structure parameter in a subprocedure

LIKEFILE(filename)

The LIKEFILE keyword is used to define a prototyped parameter as a file with the same characteristics as the filename parameter.

Note: In the following discussion, the term file parameter is used for the parameter within the procedure that was defined using the LIKEFILE keyword, the term parent file is used for the parameter of the LIKEFILE keyword whose definition is used to derive the definition of the parameter, and the term passed file is used for the file that is passed to the procedure by the caller.

Rules for the LIKEFILE keyword for prototyped parameters:

- The filename parameter of the LIKEFILE keyword must be a file that has been previously defined on a File specification.
- File specification keywords cannot be specified with the LIKEFILE keyword on a Definition specification. The file parameter uses all the settings specified by the File specification of the file specifed as the parameter of the LIKEFILE keyword.
- No other Definition keywords can be specified other than OPTIONS(*NOPASS) or OPTIONS(*OMIT).
- File parameters can be passed only between RPG programs and procedures. They are not compatible with file parameters from other programming languages, such as COBOL files, or files returned by the C fopen() or open() functions.
- A file is always passed by reference. The called procedure works directly on the same file as the calling procedure. For example, if the caller reads a record, and the called procedure updates the record and returns, the caller cannot update the record again.
- If the blocking attribute for the file cannot be determined from the File specification, the BLOCK keyword must be specified for the filename parameter.

Rules for passing and using file parameters:

- The passed file must be defined with the same parent file as the prototyped parameter.
- The file parameter is qualified. If the record formats of the parent file FILE1 are REC1 and REC2, then the record formats of the file parameter PARM must be referred to in the called procedure by PARM.REC1 and PARM.REC2.
- Any settings for the passed file that are defined using File specification keywords are in effect for all procedures that access the file, either directly or through parameter passing. For example, if the EXTFILE keyword is specified with a variable holding the external file name, and a called procedure opens the file, then the value of the caller's variable will be used to set the name of the file to be opened. If the called procedure needs to change or access those variables associated with the file through keywords, the calling procedure must pass the variables as separate parameters.
- The file-feedback built-in functions %EOF(filename), %EQUAL(filename), %FOUND(filename), %OPEN(filename), and %STATUS(filename) can be used in the called procedure to determine the current state of the file parameter by specifying the name of the file parameter as the operand to the built-in function.

  For more information on passing a file parameter between modules, see "Variables Associated with Files" on page 107 and "Example of passing a file and passing a data structure with the associated variables." on page 109.

```
 * Define a file template to be used for defining actual files
 * and the file parameter
Finfile_t  IF   E             DISK    TEMPLATE BLOCK(*YES)
F                                     EXTDESC('MYLIB/MYFILE')
F                                     RENAME(R01M2:inRec)
 * Define two actual files that can be passed to the file parameter
Ffile1                                LIKEFILE(infile_t)
F                                     EXTFILE('MYLIB/FILE1')
Ffile2                                LIKEFILE(infile_t)
F                                     EXTFILE('MYLIB/FILE2')
 * Define a data structure type for the file data
D inData_t        DS                  LIKEREC(infile_t.inRec:*INPUT)
D                                     TEMPLATE

 * Define the prototype for a procedure to handle the files
D nextValidRec    PR              N
D   infile                            LIKEFILE(infile_t)
D   data                              LIKEDS(inData_t)
 * Define variables to hold the record data
D f1Data          DS                  LIKEDS(inData_t)
D f2Data          DS                  LIKEDS(inData_t)

 /FREE
        // Process valid records from each file until one
        // of the files has no more valid records
        DOW nextValidRec(file1 : f1Data)
        AND nextValidRec(file2 : f2Data);
           // ... process the data from the files
        ENDDO;
        *INLR = '1';
 /END-FREE

 * The procedure that will process the file parameter
P nextValidRec    B
D nextValidRec    PI              N
D   infile                            LIKEFILE(infile_t)
D   data                              LIKEDS(inData_t)
 /FREE
    // Search for a valid record in the file parameter
    READ infile data;
    DOW NOT %EOF(infile);
       IF data.active = 'Y';
          RETURN *ON;            // This is a valid record
       ENDIF;
       READ infile data;
    ENDDO;
    RETURN *OFF;                 // No valid record was found
 /END-FREE
P nextValidRec    E
```

Figure 136. Passing a file as a parameter to a procedure

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 of the file on the system. *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.
- If the file is defined with the ALIAS keyword, the alias names will be used for the subfields of the data structure. Figure 113 on page 292 shows an example defining a data structure with the LIKEREC keyword where the file is defined with the ALIAS keyword.
- 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.

NOOPT

The NOOPT keyword indicates that no optimization is to be performed on the standalone field, parameter or data structure for which this keyword is specified. Specifying NOOPT ensures 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 will ensure 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.

TIP

Any data item defined in an OPM RPG/400 program is implicitly defined with NOOPT. So if you are creating a prototype for an OPM program, you should specify NOOPT for all parameters defined within the prototype. This will avoid errors for any users of the prototype.

All keywords allowed for standalone field definitions, parameters, or data structure definitions are allowed with NOOPT.

OCCURS(numeric_constant)

The OCCURS keyword allows the specification of 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 16 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 is an example showing 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                      16*
D   FLD5                          5
D DS2             DS                   OCCURS(2)
D   CHAR16                       16
D   CHR5                          5
```

Allocation of fields in storage. The occurrences of DS1 are 32 bytes apart, while the occurrences of DS2 are 21 bytes apart.

DS1 OCCURRENCE 1			DS1 OCCURRENCE 2		
POINTER	FLD5	(fill)	POINTER	FLD5	(fill)

DS2 OCCURRENCE 1		DS2 OCCURRENCE 2	
CHAR16	CHR5	CHAR16	CHR5

Figure 137. Storage Allocation of Multiple Occurrence Data Structure with Pointer Subfields

OPDESC

The OPDESC keyword specifies that operational descriptors are to be passed with the parameters that are defined within a prototype.

When OPDESC is specified, operational descriptors are passed with all character or graphic parameters that are passed by reference. If you attempt to retrieve an operational descriptor for a parameter passed by value, an error will result.

Note: Operational descriptors are not passed for UCS-2 fields.

Using CALLP with a prototyped procedure whose prototype contains OPDESC is the same as calling a procedure using CALLB (D). Operational descriptors are also passed for procedures called within expressions.

The keyword applies both to a prototype definition and to a procedure-interface definition. It cannot be used with the EXTPGM keyword.

Note: If you use the OPDESC keyword for your own procedures, the RTNPARM keyword can affect the way you call APIs such as CEEDOD to get information about your parameters. See "RTNPARM" on page 363 and "%PARMNUM (Return Parameter Number)" on page 565 for more information.

For an example of the OPDESC keyword, see the service program example in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

OPTIONS(*NOPASS *OMIT *VARSIZE *STRING *TRIM *RIGHTADJ *NULLIND)

The OPTIONS keyword is used to specify one or more parameter passing options:

- Whether a parameter must be passed
- 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.
- Whether the null-byte-map should be passed with the parameter.

When OPTIONS(*NOPASS) is specified on a definition specification, the parameter does not have to be passed on the call. Any parameters following that specification must also have *NOPASS specified. When the parameter is not passed to a program or procedure, the called program or procedure will simply function as if the parameter list did not include that parameter. If the unpassed parameter is accessed in the called program or procedure, unpredictable results will occur.

When OPTIONS(*OMIT) is specified, then the value *OMIT is allowed for that parameter. *OMIT is only allowed for CONST parameters and parameters which are passed by reference. For more information on omitted parameters, see the chapter on calling programs and procedures in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

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 *NOPASS, *OMIT, and *VARSIZE, it is up to the programmer of the procedure to ensure that these options are handled. For example, if OPTIONS(*NOPASS) is coded and you choose to pass the parameter, the procedure must check that the parameter was passed before it accesses it. The compiler will not do any checking for this. If you call APIs such as CEEDOD or CEETSTA to get information about a parameter that uses these options, the RTNPARM keyword can affect the way you call the APIs. See "RTNPARM" on page 363 and "%PARMNUM (Return Parameter Number)" on page 565 for more information.

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.

When OPTIONS(*NULLIND) is specified for a parameter, the null-byte map is passed with the parameter, giving the called procedure direct access to the null-byte map of the caller's parameter. Note the following rules for OPTIONS(*NULLIND).

- ALWNULL(*USRCTL) must be in effect.
- OPTIONS(*NULLIND) is not valid for parameters passed by value.
- The only other options that can be specified with OPTIONS(*NULLIND) are *NOPASS and *OMIT.
- Only variables may be passed as the parameter when OPTIONS(*NULLIND) is specified, and the variable must be an exact match even when CONST is specified.
- If the parameter is a data structure, the passed parameter must be defined with the same parent LIKEDS or LIKEREC as the prototyped parameter. Furthermore, the null-capability of the prototyped parameter and passed parameter must match exactly.
- A prototyped data structure parameter can have OPTIONS(*NULLIND) specified whether or not there are any null-capable subfields.
- If a non-data-structure prototyped parameter is defined with OPTIONS(*NULLIND), the parameter in the procedure interface is defined as null-capable.
- See *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for information about using OPTIONS(*NULLIND) when the calling procedure or called procedure is not written using ILE RPG.

You can specify more than one option. For example, to specify that an optional parameter can be shorter than the prototype indicates, you would code OPTIONS(*VARSIZE : *NOPASS).

The following example shows how to code a prototype and procedure that use OPTIONS(*NOPASS) to indicate that a parameter is optional.

```
 * The following prototype describes a procedure that expects
 * either one or two parameters.
D FormatAddress   PR            45A
D   City                        20A   CONST
D   Province                    20A   CONST OPTIONS(*NOPASS)
 * The first call to FormatAddress only passes one parameter.  The
 * second call passes both parameters.
C                   EVAL      A = FormatAddress('North York')
C                   EVAL      A = FormatAddress('Victoria' : 'B.C.')
C                   RETURN
 *-----------------------------------------------------------------
 * FormatAddress:
 * This procedure must check the number of parameters since the
 * second was defined with OPTIONS(*NOPASS).
 * It should only use the second parameter if it was passed.
 *-----------------------------------------------------------------
P FormatAddress   B
D FormatAddress   PI            45A
D   City                        20A   CONST
D   ProvParm                    20A   CONST OPTIONS(*NOPASS)
D Province        S             20A   INZ('Ontario')
 * Set the local variable Province to the value of the second
 * parameter if it was passed.  Otherwise let it default to
 * 'Ontario' as it was initialized.
C                   IF        %PARMS > 1
C                   EVAL      Province = ProvParm
C                   ENDIF
 * Return the city and province in the form City, Province
 * for example 'North York, Ontario'
C                   RETURN    %TRIMR(City) + ',' + Province
P FormatAddress   E
```

*Figure 138. Using OPTIONS(*NOPASS) to Indicate that a Parameter is Optional*

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.

```
FQSYSPRT   O    F   10        PRINTER USROPN
 * The following prototype describes a procedure that allows
 * the special value *OMIT to be passed as a parameter.
 * If the parameter is passed, it is set to '1' if an error
 * occurred, and '0' otherwise.
D OpenFile        PR
D   Error                        1A   OPTIONS(*OMIT)
C                   SETOFF                                       10
 * The first call to OpenFile assumes that no error will occur,
 * so it does not bother with the error code and passes *OMIT.
C                   CALLP     OpenFile(*OMIT)
 * The second call to OpenFile passes an indicator so that
 * it can check whether an error occurred.
C                   CALLP     OpenFile(*IN10)
C                   IF        *IN10
C                   ... an error occurred
C                   ENDIF
C                   RETURN
 *----------------------------------------------------------------
 * OpenFile
 * This procedure must check the number of parameters since the
 * second was defined with OPTIONS(*OMIT).
 * It should only use the second parameter if it was passed.
 *----------------------------------------------------------------
P OpenFile        B
D OpenFile        PI
D   Error                        1A   OPTIONS(*OMIT)
D SaveIn01        S              1A
 * Save the current value of indicator 01 in case it is being
 * used elsewhere.
C                   EVAL      SaveIn01 = *IN01
 * Open the file.  *IN01 will indicate if an error occurs.
C                   OPEN      QSYSPRT                              01
 * If the Error parameter was passed, update it with the indicator
C                   IF        %ADDR(Error) <> *NULL
C                   EVAL      Error = *IN01
C                   ENDIF
 * Restore *IN01 to its original value.
C                   EVAL      *IN01 = SaveIn01
P OpenFile        E
```

*Figure 139. Using OPTIONS(*OMIT)*

The following example shows how to code a prototype and procedure allowing variable-length parameters, using OPTIONS(*VARSIZE).

```
 * The following prototype describes a procedure that allows
 * both a variable-length array and a variable-length character
 * 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 Arr1            S              1A   DIM(7) CTDATA PERRCD(7)
D Arr2            S             10A   DIM(3) CTDATA
D Elem            S              5U 0
 * Call Search to search an array of 7 elements of length 1 with
 * a search argument of length 1.  Since the '*' is in the 5th
 * element of the array, Elem will have the value 5.
C                   EVAL      Elem = Search(Arr1 :
C                                        %SIZE(Arr1) : %ELEM(Arr1) :
C                                        '*' : 1)
 * Call Search to search an array of 3 elements of length 10 with
 * a search argument of length 4.  Since 'Pink' is not in the
 * array, Elem will have the value 0.
C                   EVAL      Elem = Search(Arr2 :
C                                        %SIZE(Arr2) : %ELEM(Arr2) :
C                                        'Pink' : 4)
C                   RETURN
```

*Figure 140. Using OPTIONS(*VARSIZE)* (Part 1 of 2)

```
 *-------------------------------------------------------------
 * Search:
 *   Searches for SearchFor in the array SearchIn.  Returns
 *   the element where the value is found, or 0 if not found.
 *   The character parameters can be of any length or
 *   dimension since OPTIONS(*VARSIZE) is specified for both.
 *-------------------------------------------------------------
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
 * Check each element of the array to see if it the same
 * as the SearchFor.  Use the dimension that was passed as
 * a parameter rather than the declared dimension.  Use
 * %SUBST with the length parameter since the parameters may
 * not have the declared length.
C     1           DO        ArrayDim      I                 5 0
 * 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
 * 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 140. 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.

```
 * The following prototype describes a procedure that expects
 * a null-terminated string parameter.  It returns the length
 * of the string.
D StringLen       PR             5U 0
D   Pointer                        *   VALUE OPTIONS(*STRING)
D P               S                *
D Len             S              5U 0
 * Call StringLen with a character literal.  The result will be
 * 4 since the literal is 4 bytes long.
C                   EVAL      Len = StringLen('abcd')
 * Call StringLen with a pointer to a string.  Use ALLOC to get
 * storage for the pointer, and use %STR to initialize the storage
 * to 'My string¬' where '¬' represents the null-termination
 * character x'00'.
 * 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)
 * Free the storage.
C                   DEALLOC                 P
C                   RETURN
 *-------------------------------------------------------------
 * StringLen:
 *   Returns the length of the string that the parameter is
 *   pointing to.
 *-------------------------------------------------------------
P StringLen       B
D StringLen       PI             5U 0
D   Pointer                        *   VALUE OPTIONS(*STRING)
C                   RETURN    %LEN(%STR(Pointer))
P StringLen       E
```

*Figure 141. 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
```

*Figure 142. Using OPTIONS(*TRIM)* (Part 1 of 2)

```
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 142. Using OPTIONS(*TRIM) (Part 2 of 2)*

```
 *-----------------------------------
 * DDS for file NULLFILE
 *-----------------------------------
A          R TESTREC
A            NULL1         10A         ALWNULL
A            NOTNULL2      10A
A            NULL3         10A         ALWNULL


 *-----------------------------------
 * Calling procedure
 *-----------------------------------

 * The externally-described data structure DS, and the
 * data structure DS2 defined LIKEDS(ds) have
 * null-capable fields NULL1 and NULL3.
D ds            E DS                  EXTNAME(nullFile)
D ds2             DS                  LIKEDS(ds)
 * Procedure PROC specifies OPTIONS(*NULLIND) for all its
 * parameters.  When the procedure is called, the
 * null-byte maps of the calling procedure's parameters
 * will be passed to the called procedure allowing the
 * called procedure to use %NULLIND(parmname) to access the
 * null-byte map.
D proc            PR
D   parm                              LIKEDS(ds)
D                                     OPTIONS(*NULLIND)
D   parm2                        10A   OPTIONS(*NULLIND)
D   parm3                        10A   OPTIONS(*NULLIND) CONST

 /free
     // The calling procedure sets some values
     // in the parameters and their null indicators
     %nullind(ds.null1) = *on;
     ds.notnull2 = 'abcde';
     ds.null3 = 'fghij';
     %nullind(ds.null3) = *off;
     ds2.null1 = 'abcde';
     %nullind(ds2.null1) = *on;
     %nullind(ds3.null3) = *off;
     // The procedure is called (see the code for
     // the procedure below
     proc (ds : ds2.null1 : ds2.null3);

     // After "proc" returns, the calling procedure
     // displays some results showing that the
     // called procedure changed the values of
     // the calling procedure's parameters and
     // their null-indicators
     dsply (%nullind(ds.null1));  // displays '0'
     dsply ds2.null2;             // displays 'newval'
     dsply (%nullind(ds2.null2)); // displays '0'
 /end-free
```

*Figure 143. Using OPTIONS(*NULLIND)* (Part 1 of 2)

```
 *-----------------------------------
 * Called procedure PROC
 *-----------------------------------
P                 B
D proc            PI
D   parm                               LIKEDS(ds)
D                                      OPTIONS(*NULLIND)
D   parm2                        10A   OPTIONS(*NULLIND)
D   parm3                        10A   OPTIONS(*NULLIND) CONST
 /free
     if %NULLIND(parm.null1);
        // This code will be executed because the
        // caller set on the null indicator for
        // subfield NULL1 of the parameter DS
     endif;

     if %NULLIND(parm3);
        // PARM3 is defined as null-capable since it was
        // defined with OPTIONS(*NULLIND).
        // This code will not be executed, because the
        // caller set off the null-indicator for the parameter
     endif;

     // Change some data values and null-indicator values
     // The calling procedure will see the updated values.
     parm2 = 'newvalue';
     %NULLIND(parm2) = *OFF;
     %NULLIND(parm.null1) = *OFF;
     parm.null1 = 'newval';
     return;
 /end-free
P                 E
```

*Figure 143. Using OPTIONS(*NULLIND) (Part 2 of 2)*

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 keyword OVERLAY:

1. 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.
2. 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)
```

3. 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.
4. The OVERLAY keyword is not allowed when the From-Position entry is not blank.
5. If the name parameter is a subfield, the subfield being defined must be contained completely within the subfield specified by the name parameter.
6. Alignment of subfields defined using the OVERLAY keyword must be done manually. If they are not correctly aligned, a warning message is issued.
7. 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 overlaid array, the second element of this array overlays the second element of the overlaid array, and so on. No array keywords may be specified for the subfield with the OVERLAY keyword in this situation. (Refer to Figure 144) See also "SORTA (Sort an Array)" on page 815.

   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.
8. 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 16-byte boundary.
9. 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 144. 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 145. 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 146. 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 allows you to specify the number of elements per record for a compile-time or a prerun-time 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. To remove characters from the beginning of every name, specify an empty string as the first parameter: PREFIX('':number_to_remove).

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 'nbr_of_char_replaced' parameter can be a numeric literal, a built-in function that returns a numeric value, or a numeric constant. If it is a named constant, then the constant must be defined prior to the specification containing the PREFIX keyword. In addition, if it is a built-in function, all parameters to the built-in function must be defined prior to the specification containing the keyword PREFIX.

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.

See the ALIAS keyword for information on how the PREFIX keyword interacts with the ALIAS keyword.

The following example uses PREFIX('':2) on the externally-described data structures DS1 and DS2. The fields of the file FILE1 all begin with the characters X4, and the fields of the file FILE2 all begin with the characters WR. If the two files have any fields whose names are the same aside from the initial two characters, then by specifying PREFIX('':2) for the externally-described data structures, the subfields will have identical names within the RPG program. This will enable the subfields to be assigned using the EVAL-CORR operation.

```
Ffile1     if   e             disk
Ffile2     o    e             disk
D ds1           e ds                  extname(file1) prefix('':2)
D                                     qualified
D ds2           e ds                  extname(file2) prefix('':2)
D                                     qualified
 /free
     read file1 ds1;      // Read into data structure
     eval-corr ds2 = ds1; // Assign fields with same name
     write file2 ds2;     // Write from data structure
 /end-free
```

Figure 147. Using PREFIX to remove characters from the names

For more examples, see "PREFIX(prefix{:nbr_of_char_replaced})" on page 304.

PROCPTR

The PROCPTR keyword defines an item as a procedure pointer. The internal Data-Type field (position 40) must contain a *.

See "EXTPROC({*CL|*CWIDEN|*CNOWIDEN| {*JAVA:class-name:}}name)" on page 332 for information on how to use a procedure pointer to call a procedure.

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
 Ffile2     if   e             disk

 D fileStatus      ds                  qualified
 D   file1                            N
 D   file2                            N

 C                   open(e)   file1
 C                   eval      fileStatus.file1 = %error
```

RTNPARM

The RTNPARM keyword specifies that the return value of a procedure is to be handled internally as a parameter of the same type as the defined returned value, passed by reference.

Using RTNPARM may improve performance when returning large values.

The impact on performance due to the RTNPARM keyword will vary from having a small negative impact to having a large positive impact. There may be a small negative impact when the prototyped return value is relatively small, such as an integer, or a small data structure. There will be some improvement when the prototyped return value is a larger value such as a 32767 byte data structure. The performance improvement is most apparent when the prototyped return value is a large varying length string, and the actual returned value is relatively small; for example, the prototype defines the return value as a one million byte varying length character string, and the value 'abc' is returned.

Using RTNPARM for a procedure prototype may also reduce the amount of automatic storage required for other procedures that contain calls to that procedure. For example, if procedure MYCALLER contains a call to procedure MYPROC that returns a large value, procedure MYCALLER will require additional automatic storage (even if MYCALLER does not actually call procedure MYPROC at run time). In some cases, procedure MYCALLER will not compile due to excessive automatic storage requirements; in other cases, MYCALLER is not able to be called because the total automatic storage on the call stack would exceed the maximum. Using RTNPARM avoids this problem with additional automatic storage.

Notes:
1. The additional parameter is passed as the first parameter.
2. The %PARMS and %PARMNUM built-in functions include the additional parameter in the parameter count. When the RTNPARM keyword is specified, the value returned by %PARMNUM will be one higher than the apparent parameter number.
3. When calling APIs that require a parameter number, such as CEEDOD or CEETSTA, you must account for the extra first parameter. For example, if your procedure has three parameters, and you want to find the length of the third parameter as it appears in your parameter list, you must ask for information about the fourth parameter. If you use the %PARMNUM built-in function to return the correct parameter number for calling these APIs, you do not need to worry about manually determining the correct parameter number.
4. When the calling procedure is written in a language other than RPG, the caller must code the call as though the procedure has no return value, and as though there is an additional first parameter passed by reference with the same type as the RPG return value.
5. Similarly, when the called procedure is written in a language other than RPG, the procedure must be coded without a return value, and having an additional first parameter passed by reference with the same type as the RPG return value.
6. When RTNPARM is specified for the procedure, the maximum number of prototyped parameters is 398.
7. The RTNPARM keyword is not allowed for a Java method call.

The RTNPARM keyword applies both to a prototype definition and to a procedure-interface definition.

```
1. The prototype for the procedure
D center          pr        100000a   varying
D                                     rtnparm
D   text                     50000a   const varying
D   len                         10i 0 value


2. Calling the procedure
D title           s            100a   varying
 /free
   title = center ('Chapter 1' : 20);
   // title = '     Chapter 1     '

3.The procedure
P center          b                   export
D center          pi        100000a   varying
D                                     rtnparm
D   text                     50000a   const varying
D   len                         10i 0 value
D blanks          s          50000a   inz(*blanks)
D numBlanks       s             10i 0
D startBlanks     s             10i 0
D endBlanks       s             10i 0
 /free
   if len < %len(text);
      ... handle invalid input
   endif;
   numBlanks = len - %len(text);
   startBlanks = numBlanks / 2;
   endBlanks = numBlanks - startBlanks;
   return %subst(blanks : 1 : startBlanks)
        + text
        + %subst(blanks : 1 : endBlanks);
 /end-free
P center          e
```

Figure 148. Example of a procedure with the RTNPARM keyword

```
D proc            pi              a   len(16773100) varying
D                                     rtnparm opdesc
D   p1                          10a
D   p2                          10a   options(*varsize)
D   p3                          10a   options(*omit : *nopass)

D num_parms       s             10i 0
D parm_len        s             10i 0
D desc_type       s             10i 0
D data_type       s             10i 0
D desc_info1      s             10i 0
D desc_info2      s             10i 0
D CEEDOD          pr
D   parm_num                    10i 0 const
D   desc_type                   10i 0
D   data_type                   10i 0
D   desc_info1                  10i 0
D   desc_info2                  10i 0
D   parm_len                    10i 0
D   feedback                    12a   options(*omit)
 /free
   // Get information about parameter p2
   callp CEEDOD(%parmnum(p2) : desc_type : data_type
              : desc_info1 : desc_info2
              : parm_len : *omit);
   if parm_len < 10;
      // The parameter passed for p2 is shorter than 10
   endif;

   // Find out the number of parameters passed
   num_parms = %parms();
   // If all three parameters were passed, num_parms = 4

   // test if p3 was passed
   if num_parms >= %parmnum(p3);
      // Parameter p3 was passed
      if %addr(p3) <> *null;
         // Parameter p3 was not omitted
      endif;
   endif;
```

Figure 149. Using %PARMS and calling CEEDOD with the RTNPARM keyword

```
1. The RPG prototype

D myproc          pr           200a   rtnparm
D   name                        10a   const

2. A CL module calling this RPG procedure

dcl &retval type(*char) len(200)

callprc myproc parm(&retval 'Jack Smith')
```

Figure 150. Calling a procedure with the RTNPARM keyword from another language

```
1. CL procedure GETLIBTEXT

PGM PARM(&retText &lib)

DCL &retText type(*char) len(50)
DCL &lib     type(*char) len(10)

/* Set &retText to the library text */
rtvobjd obj(&lib) objtype(*lib) text(&retText)
return

2. RPG procedure calling this CL procedure using the RTNPARM keyword

D getLibText      pr            50a   rtnparm
D   name                        10a   const
 /free
      if getLibText('MYLIB') = *blanks;
          ...
```

Figure 151. Calling a procedure with the RTNPARM keyword written in another language

STATIC{(*ALLTHREAD)}

The STATIC keyword is used:

- To specify that a local variable is stored in static storage
- To specify that the same copy of a static variable will be available to all threads in a multithreaded environment
- 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 program or service program it is contained in is first activated. It is *not* reinitialized again, even if reinitialization occurs for global definitions as part of normal cycle processing.

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 any variable in a module where THREAD(*CONCURRENT) is specified on the Control specification, STATIC(*ALLTHREAD) specifies that the same instance of a static variable will be used by all threads. If *ALLTHREAD is not specified for a static variable in a thread-concurrent module, then the variable will be in thread-local storage, meaning that each thread will have its own instance of the variable.

The following rules apply to the use of the STATIC(*ALLTHREAD) keyword:

- STATIC(*ALLTHREAD) is not allowed unless THREAD(*CONCURRENT) is specified on the Control specification.
- The STATIC keyword is implied for global variables. The STATIC keyword cannot be specified for a global variable unless *ALLTHREAD is specified as a parameter.

- A variable defined with STATIC(*ALLTHREAD) cannot be initialized to the address of variables which are not also defined with STATIC(*ALLTHREAD).

Caution: It is up to you to ensure that a static variable used in all threads is handled in a thread-safe manner. See the "Multithreading Considerations" section in the *Rational Development Studio for i: ILE RPG Programmer's Guide*, and .

Tip: It is a good idea to have a naming convention for your all-thread static variables to alert maintenance programmers and code reviewers that the variables need special handling. For example, you could add the prefix ATS_ to all your variable names that are defined with STATIC(*ALLTHREAD).

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. You must code the STATIC keyword for your prototype if and only if the Java method has the "static" attribute. The *ALLTHREAD parameter is not allowed when the STATIC keyword is specified for a prototype.

Additional Considerations for STATIC(*ALLTHREAD)

Null-capable fields: The internal variable used to hold the null indicator for a STATIC(*ALLTHREAD) null-capable field will also be defined as STATIC(*ALLTHREAD). A change to the value of the null indicator for a variable by one thread will be visible to all threads. Access to the null indicator value will not be synchronized.

Tables and Multiple-Occurrence Data Structures: The internal variable used to hold the current occurrence for a table or multiple-occurrence data structure defined with STATIC(*ALLTHREAD) will be defined in thread-local storage. Each thread will have its own instance of the current-occurrence variable.

TEMPLATE

The TEMPLATE keyword indicates that the definition is to be used only for further LIKE or LIKEDS definitions. The TEMPLATE keyword is valid for Data Structure definitions and Standalone field definitions.

Rules for the TEMPLATE keyword for Definition specifications:

1. When the TEMPLATE keyword is specified for a definition, the template name and the subfields of the template name can be used only in the following ways
   - As a parameter for the LIKE keyword
   - As a parameter for the LIKEDS keyword, if the template is a data structure
   - As a parameter for the %SIZE builtin function
   - As a parameter for the %ELEM builtin function
   - As a parameter for the %LEN builtin function in Definition specifications (for example, as a named constant or initialization value)
   - As a parameter for the %DECPOS builtin function in Definition specifications (for example, as a named constant or initialization value)
2. The INZ keyword is allowed for template data structures. This allows you to set an initialization value to be used with LIKEDS definitions of the template, through the INZ(*LIKEDS) keyword.

```
 * Define a template for the type of a NAME
D standardName    S            100A   VARYING TEMPLATE

 * Define a template for the type of an EMPLOYEE
D employee_type   DS                  QUALIFIED TEMPLATE INZ
D   name                              LIKE(standardName)
D                                     INZ('** UNKNOWN **')
D   idNum                       10I 0 INZ(0)
D   type                         1A   INZ('R')
D   years                        5I 0 INZ(-1)
 * Define a variable like the employee type, initialized
 * with the default value of the employee type
D employee        DS                  LIKEDS(employee_type)
D                                     INZ(*LIKEDS)

 * Define prototypes using the template definitions
 *
 * The "id" parameter is defined like a subfield of a
 * template data structure.
D getName         PR                  LIKE(standardName)
D   idNum
D findEmp         PR              N
D   emp                               LIKEDS(employee_type)
D   id                                LIKE(employee_type.idNum)
D                                     CONST
```

Figure 152. : Examples of TEMPLATE definitions

TIMFMT(format{separator})

The TIMFMT keyword allows the specification of 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 will be 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 36 on page 209 for valid formats and separators. For more information on internal formats, see "Internal and External Formats" on page 179.

TOFILE(file_name)

The TOFILE keyword allows the specification of a target file to which a prerun-time 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 output or 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 assigned to an output file, it is automatically written if the LR indicator is on at program termination. The array or table is written after all other records are written to the file.

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.

The VALUE keyword cannot be specified for a parameter if its prototype was defined using the EXTPGM keyword. Calls to programs require that parameters be passed by reference.

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 676 for more information.

VARYING{(2 | 4)}

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.

| The parameter of the VARYING keyword indicates the number of bytes used to
| store the current length of the variable-length item. If you specify VARYING
| without a parameter, a size of 2 is assumed if the specified length is between 1 and
| 65535; otherwise, a size of 4 is assumed. You can specify any form of the
| VARYING keyword for definitions whose length is between 1 and 65535. The
| VARYING(2) keyword cannot be specified for definitions whose length is greater
| than 65535 since 4 bytes are required to store the length.

For more information, see "Variable-Length Character, Graphic and UCS-2 Formats" on page 185.

Summary According to Definition Specification Type

Table 51 lists the required and allowed entries for each definition specification type.

Table 52 on page 371 and Table 53 on page 372 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 51. 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

Table 51. Required/Allowed Entries for each Definition Specification Type (continued)

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 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 52. Data Structure, Standalone Fields, and Named Constants Keywords

Keyword	Data Structure	Data Structure Subfield	External Subfield	Standalone Field	Named Constant
ALIGN	A				
ALT		A	A	A	
ALTSEQ	A	A	A	A	
ASCEND		A	A	A	
BASED	A			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	
EXPORT[2]	A			A	
EXTFLD			A		
EXTFMT		A	A	A	
EXTNAME[4]	A				
FROMFILE[2]		A	A	A	
IMPORT[2]	A			A	

Table 52. Data Structure, Standalone Fields, and Named Constants Keywords (continued)

Keyword	Data Structure	Data Structure Subfield	External Subfield	Standalone Field	Named Constant
INZ	A	A	A	A	
LEN	A	A		A	
LIKE		A		A	
LIKEDS[5]	A	A			
LIKEREC	A	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	
TEMPLATE	A			A	
TIMFMT		A		A	
TOFILE[2]		A	A	A	
VARYING		A		A	

Notes:

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 applies only to global definitions.
3. This keyword applies only to local definitions.
4. This keyword applies only to externally described data structures.
5. This keyword applies only to program-described data structures.

Table 53. Prototype, Procedure Interface, and Parameter Keywords

Keyword	Prototype (PR)	Procedure Interface (PI)	PR or PI Parameter
ALTSEQ	A	A	A
ASCEND			A
CCSID	A	A	A
CLASS	A	A	A
CONST			A
DATFMT	A	A	A
DESCEND			A
DIM	A	A	A
EXTPGM	A	A	
EXTPROC	A	A	

Table 53. Prototype, Procedure Interface, and Parameter Keywords (continued)

Keyword	Prototype (PR)	Procedure Interface (PI)	PR or PI Parameter
LEN	A	A	A
LIKE	A	A	A
LIKEFILE			A
LIKEDS	A	A	A
LIKEREC	A	A	A
NOOPT			A
OPDESC	A	A	
OPTIONS			A
PROCPTR	A	A	A
RTNPARM	A	A	
STATIC	A	A	
TIMFMT	A	A	A
VALUE			A
VARYING	A	A	A

Chapter 15. Input Specifications

For a program-described input file, input specifications describe the types of records within the file, the sequence of the types of records, the fields within a record, the data within the field, indicators based on the contents of the fields, control fields, fields used for matching records, and fields used for sequence checking. For an externally described file, input specifications are optional and can be used to add RPG IV functions to the external description.

Input specifications are not used for all of the files in your program. For some files, you must code data structures in the result field of your input operatons. The following files in your program do not use Input specifications:

- Files defined in subprocedures
- Files defined with the QUALIFIED keyword
- Files defined with the TEMPLATE keyword
- Files defined with the LIKEFILE keyword

Detailed information for the input specifications is given in:

- Entries for program described files
- Entries for externally described files

Input Specification Statement

The general layout for the Input specification is as follows:

- 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++SqNORiPos1+NCCPos2+NCCPos3+NCC.................................Comments++++++++++++
I.........And..RiPos1+NCCPos2+NCCPos3+NCC.................................Comments++++++++++++
```

Figure 153. 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+++++++++L1M1FrPlMnZr......Comments++++++++++++
```

Figure 154. 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 RPG IV functions are to be added.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
IRcdname+++....Ri.....................................................Comments++++++++++++
```

Figure 155. Externally Described Record Layout

- Field description entries (positions 21 through 30, 49 through 66, and 69 through 74), which describe the RPG IV 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+++++++++L1M1..PlMnZr......Comments++++++++++++
```

Figure 156. Externally Described Field Layout

Program Described Files

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
Same file name that appears on the file description specifications for the input file.

Enter the name of the file to be described in these positions. This name must be the same 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 identification line for each file and can be entered on subsequent record identification 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 381 and "OR Relationship" on page 381.

Positions 17-18 (Sequence)

Entry **Explanation**

Any two alphabetic characters
The program does not check for special sequence.

Any two-digit number
The program checks for special sequence within the group.

The numeric sequence entry combined with the number (position 19) and option (position 20) entries causes the program to check the sequence of input records within a file. If the sequence is not correct, control passes to the RPG IV exception/error handling routine. If AND or OR lines are specified, the sequence entry is made on the main record line of the group, not on the AND or OR lines.

Alphabetic and numeric entries can be made for different records (different record identification lines) in the same file, but records with alphabetic entries must be specified before records with numeric entries.

Alphabetic Entries

Enter any two alphabetic characters in these positions when no sequence checking is to be done. It is common programming practice to specify these codes in a sequence that aids in program documentation. However, it is not necessary to use unique alphabetic entries.

Numeric Entries

Enter a unique numeric code in positions 17 and 18 if one record type must be read before another record type in a file. Numeric entries must be in ascending order, starting with 01, but need not be consecutive. When a numeric entry is used, the appropriate entries must be made in positions 19 and 20.

To specify sequence checking, each record type must have a record identification code, and the record types must be numbered in the order in which they should appear. This order is checked as the records are read. If a record type is out of sequence, control passes to the RPG IV exception/error handling routine.

Sequence numbers ensure only that all records of each record type precede the records of higher sequence-numbered record types. The sequence numbers do not ensure that records within a record type are in any certain order. Sequence numbers are unrelated to control levels and do not provide for checking data in fields of a record for a special sequence. Use positions 65 and 66 (matching fields) to indicate that data in fields of a record should be checked for a special sequence.

Position 19 (Number)

Entry **Explanation**

Blank The program does not check record types for a special sequence (positions 17 and 18 have alphabetic entries).

1 Only one record of this type can be present in the sequenced group.

N One or more records of this type can be present in the sequenced group.

This entry must be used when a numeric entry is made in positions 17 and 18. If an alphabetic entry is made in positions 17 and 18, this entry must be blank.

Position 20 (Option)

Entry Explanation

Blank The record type must be present if sequence checking is specified.

O The record type is optional (that is, it may or may not be present) if sequence checking is specified.

This entry must be blank if positions 17 and 18 contain an alphabetic entry.

Sequence checking of record types has no meaning when all record types within a file are specified as optional (alphabetic entry in positions 17 and 18 or O entry in position 20).

Positions 21-22 (Record Identifying Indicator, or **)

Entry Explanation

Blank No indicator is used.

01-99 General indicator.

L1-L9 or LR
Control level indicator used for a record identifying indicator.

H1-H9 Halt indicator.

U1-U8 External indicator.

RT Return indicator.

** Lookahead record (not an indicator). Lookahead can be used only with a primary or secondary file.

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. The normal entry is one of the indicators 01 to 99; however, the control level indicators L1 through L9 and LR can be used to cause certain total steps to be processed. If a control level indicator is specified, lower control level indicators are not set on. The halt indicators H1 through H9 can be used to stop processing. The return indicator (RT) is used to return to the calling program.

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. However, at the end of the cycle, all record identifying indicators are set off before another record is selected.

Lookahead Fields

The entry of ** is used for the lookahead function. This function lets you look at information in the next record in a file. You can look not only at the file currently selected for processing but also at other files present but not selected during this cycle.

Field description lines must contain From and To entries in the record, a field name, and decimal positions if the field is numeric. Note that a lookahead field may not be specified as a field name on input specifications or as a data structure name on definition specifications or as a Result Field on Calculation Specifications.

Positions 17 and 18 must contain an alphabetic entry. The lookahead fields are defined in positions 49 through 62 of the lines following the line containing ** in positions 21 and 22. Positions 63 through 80 must be blank.

Any or all of the fields in a record can be defined as lookahead fields. This definition applies to all records in the file, regardless of their type. If a field is used both as a lookahead field and as a normal input field, it must be defined twice with different names.

The lookahead function can be specified only for primary and secondary files and can be specified only once for a file. It cannot be used for full procedural files (identified by an F in position 18 of the file description specifications), or with AND or OR lines.

When a record is being processed from a combined file or an update file, the data in the lookahead field is the same as the data in the record being processed, not the data in the next record.

The lookahead function causes information in the file information data structure to be updated with data pertaining to the lookahead record, not to the current primary record.

If an array element is specified as a lookahead field, the entire array is classified as a lookahead field.

So that the end of the file can be recognized, lookahead fields are filled with a special value when all records in the file have been processed. For character fields, this value is all '9's; for all other data types, this value is the same as *HIVAL.

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 or 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 need not 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 RPG IVexception/error handling routine.

Positions 23-27, 31-35, and 39-43 (Position)

Entry **Explanation**

Blank No record identification code is present.

1-32766
The position that contains the record identification code in the record.

In these positions enter the position that contains the record identification code in each 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 Record identification code must not be present.

Enter an N in this position if the code described must not be present in the specified record position.

Positions 29, 37, and 45 (Code Part)

Entry **Explanation**

C Entire character

Z Zone portion of character

D Digit portion of character.

This entry specifies what part of the character in the record identification code is to be tested.

Character (C): The C entry indicates that the complete structure (zone and digit) of the character is to be tested.

Zone (Z): The Z entry indicates that the zone portion of the character is to be tested. The zone entry causes the four high-order bits of the character entry to be compared with the zone portion of the character in the record position specified in the position entry. The following three special cases are exceptions:

- The hexadecimal representation of an & (ampersand) is 50. However, when an ampersand is coded in the character entry, it is treated as if its hexadecimal representation were C0, that is, as if it had the same zone as A through I. An ampersand in the input data satisfies two zone checks: one for a hexadecimal 5 zone, the other for a hexadecimal C zone.
- The hexadecimal representation of a - (minus sign) is 60. However, when a minus sign is coded in the character entry, it is treated as if its hexadecimal representation were D0, that is, as if it had the same zone as J through R. A minus sign in the input data satisfies two zone checks: one for a hexadecimal 6 zone, the other for a hexadecimal D zone.
- The hexadecimal representation of a blank is 40. However, when a blank is coded in the character entry, it is treated as if its hexadecimal representation were F0, that is, as if it had the same zone as 0 through 9. A blank in the input data satisfies two zone checks: one for a hexadecimal 4 zone, the other for a hexadecimal F zone.

Digit (D): The D entry indicates that the digit portion of the character is to be tested. The four low-order bits of the character are compared with the character specified by the position entry.

Positions 30, 38, and 46 (Character)

In this position enter the identifying character that is to be compared with the character in the position specified in the input record.

The check for record type always 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 Table 33 on page 207 and Table 36 on page 209 for valid date and time formats.

For character, graphic, 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 Formats” on page 185.

For more information on external formats, see “Internal and External Formats” on page 179.

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 Table 33 on page 207 and Table 36 on page 209 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)

Entry **Explanation**

Blank The input field is in zoned decimal format or is a character field.

A Character field (fixed- or variable-length format)

C UCS-2 field (fixed- or variable-length format)

G Graphic field (fixed- or variable-length format)

B Numeric field (binary format)

F Numeric field (float format)

I Numeric field (integer format)

L Numeric field with a preceding (left) plus or minus sign (zoned decimal format)

N Character field (Indicator 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 format)

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.

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 as follows:

Positions	Type of Field
63	Zoned decimal numeric (63 digits)
32	Packed numeric (63 digits)
4	Binary (9 digits)
8	Integer (20 digits)
8	Unsigned (20 digits)
8	Float (8 bytes)
64	Numeric with leading or trailing sign (63 digits)
10	Date
8	Time
26	Timestamp
32766	Character (32766 characters)
32766	Graphic or UCS-2 (16383 double-byte characters)

32766 Variable-Length Character (32764 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

0-63 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 an RPG IV program. This name must follow the rules for .

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, control level (positions 63-64), matching fields (positions 65 and 66), and 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 (Control Level)

Entry **Explanation**

Blank This field is not a control field. Control level indicators cannot be used with full procedural files.

L1-L9 This field is a control field.

Positions 63 and 64 indicate the fields that are used as control fields. A change in the contents of a control field causes all operations conditioned by that control level indicator and by all lower level indicators to be processed.

A split control field is a control field that is made up of more than one field, each having the same control level indicator. The first field specified with that control level indicator is placed in the high-order position of the split control field, and the last field specified with the same control level indicator is placed in the low-order position of the split control field.

Binary, float, integer, character varying, graphic varying, UCS-2 and unsigned fields cannot be used as control fields.

Positions 65-66 (Matching Fields)

Entry	Explanation
Blank	This field is not a match field.
M1-M9	This field is a match field.

This entry is used to match the records of one file with those of another or to sequence check match fields within one file. Match fields can be specified only for fields in primary and secondary files.

Binary, float, integer, character varying, graphic varying, UCS-2, and unsigned fields cannot be used as match fields.

Match fields within a record are designated by an M1 through M9 code entered in positions 65 and 66 of the appropriate field description specification line. A maximum of nine match fields can be specified.

The match field codes M1 through M9 can be assigned in any sequence. For example, M3 can be defined on the line before M1, or M1 need not be defined at all.

When more than one match field code is used for a record, all fields can be considered as one large field. M1 or the lowest code used is the rightmost or low-order position of the field. M9 or the highest code used is the leftmost or high-order position of the field.

The ALTSEQ (alternate collating sequence) and FTRANS (file translation) keywords on the control specification can be used to alter the collating sequence for match fields.

If match fields are specified for only a single sequential file (input, update, or combined), match fields within the file are sequence checked. The MR indicator is not set on and cannot be used in the program. An out-of-sequence record causes the RPG IV exception/error handling routine to be given control.

In addition to sequence checking, match fields are used to match records from the primary file with those from secondary files.

Positions 67-68 (Field Record Relation)

Entry **Explanation**

Blank The field is common to all record types.

01-99 General indicators.

L1-L9 Control level indicators.

MR Matching record indicator.

U1-U8 External indicators.

H1-H9 Halt indicators.

RT Return indicator.

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 by the RPG IV program 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.

Field record relation indicators can be used with control level fields (positions 63 and 64) and matching fields (positions 65 and 66).

Positions 69-74 (Field Indicators)

Entry **Explanation**

Blank No indicator specified

01-99 General indicators

H1-H9 Halt indicator

U1-U8 External indicators

RT Return indicator.

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 or look-ahead fields.

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, or 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

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 by the compiler, 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.
- Control level or matching field indicators are to be used.
- 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. (The file name cannot be used for an externally described file.)
- The RPG IV 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 as described under “Program Described Files” on page 376 earlier in this chapter, except for look-ahead specifications, which are not allowed for an externally described file.

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 control level, field indicator, and match field 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 to an RPG IV 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. If DEBUG or DEBUG(*YES) is specified, all externally described input fields will be read even if they are not used 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 304.

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 RPG IV function (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 .

Indicators are not allowed to be null-capable.

Positions 63-64 (Control Level)

This entry indicates whether the field is to be used as a control field in the program.

Entry	Explanation
Blank	This field is not a control field.

L1-L9 This field is a control field.

Null-capable and UCS-2 fields cannot be used as control fields.

Note: For externally described files, split control fields are combined in the order in which the fields are specified on the data description specifications (DDS), not in the order in which the fields are specified on the input specifications.

Positions 65-66 (Matching Fields)

This entry indicates whether the field is to be used as a match field.

Entry **Explanation**

Blank This field is not a match field.

M1-M9
This field is a match field.

Null-capable and UCS-2 fields cannot be used as matching fields.

See "Positions 65-66 (Matching Fields)" on page 385 for more information on match fields.

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

H1-H9 Halt indicators

U1-U8 External indicators

RT Return indicator.

Field indicators are allowed for null-capable fields only if the ALWNULL(*USRCTL) keyword is specified on a control specification or as a command parameter.

If a field is a null-capable field and the value is null, the indicator is set off.

See "Positions 69-74 (Field Indicators)" on page 386 for more information.

Positions 75-80 (Reserved)

Positions 75 through 80 must be blank.

Chapter 16. Calculation Specifications

Calculation specifications indicate the operations done on the data in a program.

Calculation specifications within the main source section must be grouped in the following order:

- Detail calculations
- Total calculations
- Subroutines

Calculation specifications for subprocedures include two groups:

- Body of the subprocedure
- Subroutines

Calculations within the groups must be specified in the order in which they are to be done.

Note: If the keyword MAIN or NOMAIN is specified on the control specification, then only declarative calculation specifications are allowed in the main source section.

You can specify calculation specifications in three different formats:

- "Traditional Syntax"
- "Extended Factor 2 Syntax" on page 397
- "Free-Form Syntax" on page 399.

See Chapter 22, "Operation Codes," on page 607 for details on how the calculation specification entries must be specified for individual operation codes.

The calculation specification can also be used to enter SQL statements into an ILE RPG program. See *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* and the iSeries Information Center database and file systems category for more information.

Traditional Syntax

The general layout for the calculation specification is as follows:

- The calculation specification type (C) is entered in position 6
- The non-commentary 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 control level indicator and 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 397 and Chapter 20, "Expressions," on page 477) 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. For the uses of these positions, see the individual operation codes in Chapter 22, “Operation Codes,” on page 607.

- The comments section of the specification extends from position 81 to position 100

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments++++++++++++
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 157. 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 158. 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 at detail calculation time for each program cycle 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.
L0	The calculation operation is done at total calculation time for each program cycle.
L1-L9	The calculation operation is done at total calculation time when the control level indicator is on. The indicator is set on either through a level break or as the result of an input or calculation operation.
LR	The calculation operation is done after the last record has been processed or after the LR indicator has been set on.
SR	The calculation operation is part of an RPG IV 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.

Control Level Indicators

The L0 entry is used in positions 7 and 8 to indicate that the calculation is always done during total calculation time.

If indicators L1 through L9 are specified in positions 7 and 8, the calculation is processed at total calculation time only when the specified indicator is on. Remember that, if L1 through L9 are set on by a control break, all lower level indicators are also set on. If positions 7 and 8 are blank, the calculation is done at detail time calculation, is a statement within a subroutine, is a declarative statement, or is a continuation line.

The following operations can be specified within total calculations with positions 7 and 8 blank: PLIST, PARM, KLIST, KFLD, TAG, DEFINE, and ELSE. (Conditioning indicators in positions 9 through 11 are not allowed with these operations.) In addition, all the preceding operations except TAG and ELSE can be specified anywhere within the calculations, even between an ENDSR operation of one subroutine and the BEGSR operation of the next subroutine or after the ENDSR operation for the last subroutine.

Note: Control indicators cannot be specified in subprocedures.

Last Record Indicator

The LR Indicator, if specified in positions 7 and 8, causes the calculation to be done during the last total calculation time. Note that the LR indicator cannot be specified in subprocedures.

If there is a primary file but no secondary files in the program, the LR indicator is set on after the last input record has been read, the calculations specified for the record have been done, and the detail output for the last record read has been completed.

If there is more than one input file (primary and secondary), the programmer determines which files are to be checked for end-of-file by entering an E in position 19 of the file description specifications. LR is set on when all files with an end-of-file specification have been completely read, when detail output for the last record in these files has been completed, and after all matching secondary records have been processed.

When the LR indicator is set on after the last input record has been read, all control indicators L1 through L9 defined to the program are also set on.

Subroutine Identifier

An SR entry in positions 7 and 8 may optionally be used for operations within subroutines as a documentation aid. Subroutine lines must appear after the total calculation specifications. 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. A control level indicator (L1 through L9, L0, or LR) is entered for total calculations, an SR or blanks for subroutines, and a blank for detail calculations.

Positions 9-11 (Indicators)

Entry	Explanation
Blank	The operation is processed on every record
01-99	General indicators.
KA-KN, KP-KY	Function key indicators.
L1-L9	Control level indicators.
LR	Last record indicator.
MR	Matching record indicator.
H1-H9	Halt indicators.
RT	Return indicator.
U1-U8	External indicators.
OA-OG, OV	Overflow indicator.

Positions 10 and 11 contain an indicator that is tested to determine if a particular calculation is to be processed. 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 RPG IV 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 that are valid 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 22, “Operation Codes,” on page 607. 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 19, “Operations,” on page 423 and Chapter 22, “Operation Codes,” on page 607. For further information on the operation code extenders, see “Operation Extender.”

Operation Extender

Entry	Explanation
Blank	No operation extension supplied
A	Used on the DUMP operation to indicate that the operation is always performed regardless of the DEBUG option set on the H specification.
H	Half adjust (round) result of numeric operation
N	Record is read but not locked

Set pointer to *NULL after successful DEALLOC

P Pad the result field with blanks

D Pass operational descriptors on bound call

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 all valid entries: MULT(H), MULT (H), MULT (H).

More than one operation extender can be specified. For example, the CALLP operation can specify both error handling and the default precision rules with CALLP(EM).

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.

A D when specified on the CALLB operation code indicates that operational descriptors are included.

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 486.

An M indicates that the default precision rules are used.

An 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.

An 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 22, "Operation Codes," on page 607.

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 lookahead field or a user date field. With some operation codes, two operands may be specified separated by a colon. See Chapter 22, "Operation Codes," on page 607 for the result field rules for individual operation codes.

Positions 64-68 (Field Length)

Entry	Explanation
1-63	Numeric field length.
# **1-99999**	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.

Positions 69-70 (Decimal Positions)

Entry	Explanation
Blank	The result field is character data, has been defined elsewhere in the program, or no field name has been specified.
0-63	Number of decimal positions in a numeric result field.

Positions 69-70 indicate the number of positions to the right of the decimal in a numeric result field. If the numeric result field contains no decimal positions, enter a '0' (zero). This position must be blank if the result field is character data or if no field length is specified. The number of decimal positions specified cannot exceed the length of the field.

Positions 71-76 (Resulting Indicators)

These positions can be used, for example, 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 22, "Operation Codes," on page 607 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
KA-KN, KP-KY	Function key indicators
H1-H9	Halt indicators
L1-L9	Control level indicators
LR	Last record indicator
OA-OG, OV	Overflow indicators
U1-U8	External indicators
RT	Return 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.

Remember the following points when specifying resulting indicators:

- 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.
- When a control level indicator (L1 through L9) is set on, the lower level indicators are not set on.
- When a halt indicator (H1 through H9) is set on, the program ends abnormally at the next *GETIN point in the cycle, or when a RETURN operation is processed, unless the halt indicator is set off before the indicator is tested.

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 392.

Positions 9-11 (Indicators)

See "Positions 9-11 (Indicators)" on page 394.

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 19, “Operations,” on page 423 and Chapter 22, “Operation Codes,” on page 607. For further information on the operation code extenders, see “Operation Extender.”

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 CALLP, DOU, DOW, EVAL, IF, RETURN, and WHEN operations.

Error handling may be specified, using the 'E' extender, on CALLP operations.

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 623
- “DOU (Do Until)” on page 660
- “DOW (Do While)” on page 663
- “EVAL (Evaluate expression)” on page 676
- “EVALR (Evaluate expression, right adjust)” on page 678
- “FOR (For)” on page 692
- “IF (If)” on page 698
- “ON-ERROR (On Error)” on page 758
- “RETURN (Return to Caller)” on page 795
- “WHEN (When True Then Select)” on page 843

See the specific operation codes for more information. See “Continuation Rules” on page 249 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, if no extenders are needed, and if the variable or prototype does not have the same name as an 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.)

To indicate the start of total calculations, end the free-form group and code a fixed-form calculation specification with a control level specified in positions 7-8. The total calculations may be specified using free-form calculation syntax. Since the free-form calculation specification does not include a control-level entry, calculations to be performed on specific level breaks should be conditioned using the statement "IF *INLx;".

```
*..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 159. 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 160. 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 Table 54 on page 423 for a list of the operation codes that can use free-form syntax. For operations that cannot use free-form syntax, check the detailed description in Chapter 22, "Operation Codes," on page 607 to see if there is a suggested replacement. See "Continuation Rules" on page 249 for more information on coding continuation lines.

Chapter 17. 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 MAIN or NOMAIN is coded on a control specification, only exception output can be done.

Output specifications are not used for all of the files in your program. For some files, you must code data structures in the result field of your output and update operatons. The following files in your program do not use Output specifications:

- Files defined in subprocedures
- Files defined with the QUALIFIED keyword
- Files defined with the TEMPLATE keyword
- Files defined with the LIKEFILE keyword

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). Detailed information for each category of output specifications is given in:

- Entries for program-described files
- Entries for externally described files

Output Specification Statement

The general layout for the Output specification is as follows:

- the output specification type (O) 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 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++DF..N01N02N03Excnam++++B++A++Sb+Sa+.............................Comment+++++++++++++
OFilename++DAddN01N02N03Excnam++++.........................................Comment+++++++++++++
O.........And..N01N02N03Excnam++++.........................................Comment+++++++++++++
```

Figure 161. 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 162. 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+++D...N01N02N03Excnam++++.....................................Comment+++++++++++++
ORcdname+++DAddN01N02N03Excnam++++.....................................Comment+++++++++++++
O.........And..N01N02N03Excnam++++.....................................Comment+++++++++++++
```

Figure 163. 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 164. Externally Described Field Layout

Program Described Files

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 valid file name
Same file name that appears on the file description specifications for the output file.

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.

Positions 16 through 18 specify AND/OR lines for output operations. 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. For an AND relationship and fetch overflow position 18 must be specified on the first line only (file name line). A fetch overflow entry is required on OR lines for record types requiring the fetch overflow routine.

Positions 7 through 15 must be blank when AND/OR is specified.

An unlimited number of AND/OR lines can be specified on the output specifications.

Position 17 (Type)

Entry	Explanation
H or D	Detail records usually contain data that comes directly from the input record or that is the result of calculations processed at detail time. Heading records usually contain constant identifying information such as titles, column headings, page number, and date. No distinction is made between heading and detail records. The H/D specifications are available to help the programmer document the program.
T	Total records usually contain data that is the end result of specific calculations on several detail records.
E	Exception records are written during calculation time. Exception records can be specified only when the operation code EXCEPT is used. See "EXCEPT (Calculation Time Output)" on page 684 for further information on the EXCEPT operation code.

Position 17 indicates the type of record to be written. Position 17 must have an entry for every output record. Heading (H) and detail (D) lines are both processed as detail records. No special sequence is required for coding the output records; however, lines are handled at separate times within the program cycle based on their record type. See Figure 7 on page 32 and Figure 8 on page 33 for more information on when in the cycle output is performed.

Note: If MAIN or NOMAIN is coded on a control specification, only exception output can be done.

Positions 18-20 (Record Addition/Deletion)

Entry **Explanation**

ADD Add a record to the file or subfile.

DEL Delete the last record read from the file. The deleted record cannot be retrieved; the record is deleted from the system.

An entry of ADD is valid for input, output, or update files. DEL is valid for update DISK files only. When ADD is specified, there must be an A in position 20 of the corresponding file-description specification.

If positions 18-20 are blank, then for an output file, the record will be added; for an update file, the record is updated.

The Record-Addition/Deletion entry must appear on the same line that contains the record type (H, D, T, E) specification (position 17). If an AND/OR line is used following an ADD or DEL entry, this entry applies to the AND/OR line also.

Position 18 (Fetch Overflow/Release)

This entry must be blank if the LIKEFILE keyword is specified. The File Designation of the parent file is used.

Entry **Explanation**

Blank Must be blank for all files except printer files (PRINTER specified in positions 36 through 42 of the file description specifications). If position 18 is blank for printer files, overflow is not fetched.

F Fetch overflow.

R Release a device (workstation) after output.

Fetch Overflow

An F in position 18 specifies fetch overflow for the printer file defined on this line. This file must be a printer file that has overflow lines. Fetch overflow is processed only when an overflow occurs and when all conditions specified by the indicators in positions 21 through 29 are satisfied. An overflow indicator cannot be specified on the same line as fetch overflow.

If an overflow indicator has not been specified with the OFLIND keyword on the file description specifications for a printer file, the compiler assigns one to the file. An overflow line is generated by the compiler for the file, except when no other output records exist for the file or when the printer uses externally described data. This compiler-generated overflow can be fetched.

Overflow lines can be written during detail, total, or exception output time. When the fetch overflow is specified, only overflow output associated with the file containing the processed fetch is output. The fetch overflow entry (F) is required on each OR line for record types that require the overflow routine. The fetch overflow routine does not automatically advance forms. For detailed information on the overflow routine see “Overflow Routine” on page 39 and Figure 9 on page 39

The form length and overflow line can be specified using the FORMLEN and OFLIND keywords on the file description specifications, in the printer device file, or through an IBM i override command.

Release

After an output operation is complete, the device used in the operation is released if you have specified an R in position 18 of the corresponding output specifications. See the "REL (Release)" on page 787 operation for further information on releasing devices.

Positions 21-29 (Output Conditioning Indicators)

Entry	Explanation
Blank	The line or field is output every time the record (heading, detail, total, or exception) is checked for output.
01-99	A general indicator that is used as a resulting indicator, field indicator, or record identifying indicator.
KA-KN, KP-KY	Function key indicators.
L1-L9	Control level indicators.
H1-H9	Halt indicators.
U1-U8	External indicator set before running the program or set as a result of a calculation operation.
OA-OG, OV	Overflow indicator previously assigned to this file.
MR	Matching record indicator.
LR	Last record indicator.
RT	Return indicator.
1P	First-page indicator. Valid only on heading or detail lines.

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 409 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.

For an AND relationship, fetch overflow (position 18) can only be specified on the first line. Positions 40 through 51 (spacing and skipping) must be blank for all AND lines.

An overflow indicator must be defined on the file description specifications with the OFLIND keyword before it can be used as a conditioning indicator. If a line is to be conditioned as an overflow line, the overflow indicator must appear on the main specification line or on the OR line. If an overflow indicator is used on an

AND line, the line is *not* treated as an overflow line, but the overflow indicator is checked before the line is written. In this case, the overflow indicator is treated like any other output indicator.

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. If fetch overflow (position 18) is used, it must be specified on each 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 . 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.

If an exception record with an EXCEPT name is conditioned by an overflow indicator, the record is written only during the overflow portion of the RPG IV cycle or during fetch overflow. The record is not written at the time the EXCEPT operation is processed.

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 Figure 165, the record lock in file RCDA is released by the EXCEPT operation. For more information, see *ILE Application Development Example*, SC41-5602-00.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
C*
C     KEY           CHAIN     RCDA
C                   EXCEPT    RELEASE
ORcdname+++D...N01N02N03Excnam++++.......................................
O
O*
ORCDA      E           RELEASE
O*                   (no fields)
```

Figure 165. 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:
- Skip before
- Space before
- Print a line
- Skip after
- 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
1-255	Skipping values

Field Description and Control Entries

These entries determine under what conditions and in what format fields of a record are to be written.

Each field is described on a separate line. 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 409 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 (Output Conditioning Indicators)” on page 405. Indicators used to condition field descriptions lines cannot be specified in an AND/OR relationship. Conditioning indicators cannot be specified on format name specifications (see “Positions 53-80 (Constant, Edit Word, Data Attributes, Format Name)” on page 412) for program described WORKSTN files.

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 RPG IV 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 or edit word 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 (Look Up a Table or Array Element)” on page 711 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, code PAGE in positions 30 through 43 as the name of the output field. Indicators specified in positions 21 through 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 “RPG IV Words with Special Functions/Reserved Words” on page 5.

*PLACE

*PLACE is an RPG IV reserved word that 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 coded 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

The user date reserved words (UDATE, *DATE, UDAY, *DAY, UMONTH, *MONTH, UYEAR, *YEAR) allow the programmer to supply a date for the program at run time. For more information on the user date reserved words, see “Rules for User Date” on page 8.

*IN, *INxx, *IN(xx)

The reserved words *IN, *INxx and *IN(xx) allow the programmer to refer to and manipulate RPG IV indicators as data.

Position 44 (Edit Codes)

Entry	Explanation
Blank	No edit code is used.
1-9, 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 9, A through D, J through Q, X, Y, Z, and blank.

Note: The entry must be blank if you are writing a float output field.

For more information on edit codes see Chapter 10, “Editing Numeric Fields,” on page 229.

Edit codes 5 through 9 are user-defined edit codes and are defined externally by an IBM i function. The edit code is determined at compilation time. Subsequent changes to a user-defined edit code will not affect the editing by the RPG IV compiler unless the program is recompiled.

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.

Resetting fields to zeros may be useful in total output when totals are accumulated and written for each control group in a program. After the total is accumulated and written for one control group, the total field can be reset to zeros before accumulation begins on the total for the next control group.

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
K1-K10	Length of format name for WORKSTN file.

Positions 47 through 51 define the end position of a field or constant on the output record, or define the length of the data description specifications record format name for a program described WORKSTN file.

The K identifies the entry as a length rather than an end position, and the number following the K indicates the length of the record format name. For example, if the format name is CUSPMT, the entry in positions 50 and 51 is K6. Leading zeros are permitted following the K, and the entry must be right-adjusted.

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:

```
EP = PEP +nnnn + FL
EP = PEP −nnnn + FL
```

EP is the calculated end position. PEP is the previous end position. For the first field specification in the record, PEP is equal to zero. FL 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 (PEP) is 6, the number of positions to be placed between the fields (nnnn) is 5, and the field length (FL) is 10, the end position (EP) 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

- For numeric fields the data is to be written in zoned decimal format.
- For float numeric fields, the data is to be written in the external display representation.
- For graphic fields, the data is to be written with SO/SI brackets.
- For UCS-2 fields, the data is to be written in UCS-2 format.
- For date, time, and timestamp fields the data is to be written without format conversion performed.
- For character fields, the data is to be written as it is stored.

A 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.

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 The graphic field (without SO/SI brackets) 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 to be written with a preceding (left) plus or minus sign, in zoned-decimal format.

N The character field is to be written in indicator format.

P The numeric field is to be written in packed-decimal format.

R The numeric field is to be 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 will be converted to the format specified in positions 53-80 or to the default file date format.

T Time field— the time field will be converted to the format specified in positions 53-80 or to the default file time format.

Z Valid for Timestamp fields only.

This position must be blank if editing is specified.

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.

A 'G' or blank must be specified for a graphic field in a program-described file. If 'G' is specified, then, the data will be output without SO/SI. If this column is blank for program-described output, then SO/SI brackets will be placed around the field in the output record by the compiler if the field is of type graphic. You must ensure that there is sufficient room in the output record for both the data and the SO/SI characters.

Positions 53-80 (Constant, Edit Word, Data Attributes, Format Name)

Positions 53 through 80 are used to specify a constant, an edit word, a data attribute, or a format name for a program described file.

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 249 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 Words

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 236 for details.

Edit words must be character literals or named constants. Graphic, UCS-2, or hexadecimal literals and 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 Table 33 on page 207 and Table 36 on page 209 for valid date and time formats.

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 Formats" on page 185.

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 179.

Record Format Name

The name of the data description specifications record format that is used by a program described WORKSTN file must be specified in positions 53 through 62. One format name is required for each output record for the WORKSTN file; specifying more than one format name per record is not allowed. Conditioning indicators cannot be specified on format name specifications for program described WORKSTN files. The format name must be enclosed in apostrophes. You must also enter Kn in positions 47 through 51, where n is the length of the format name. For example, if the format name is 'CUSPMT', enter K6 in positions 50 and 51. A named constant can also be used.

Externally Described Files

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 (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 403 for more information.

Position 17 (Type)

Entry	Explanation
H or D	Detail records
T	Total records
E	Exception records.

Position 17 indicates the type of record to be written. See “Position 17 (Type)” on page 403 for more information.

Position 18 (Release)

Entry	Explanation
R	Release a device after output.

See “Release” on page 405 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.

For more information on record addition, see “Positions 18-20 (Record Addition/Deletion)” on page 403.

Positions 21-29 (Output Indicators)

Output indicators for externally described files are specified in the same way as those for program described files. The overflow indicators OA-OG, OV are not valid for externally described files. For more information on output indicators, see "Positions 21-29 (Output Conditioning Indicators)" on page 405.

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 406 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 (Output Conditioning Indicators)" on page 405 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.

Resetting fields to zeros may be useful in total output when totals are accumulated and written for each control group in a program. After the total is accumulated and written for one control group, the total field can be reset to zeros before accumulation begins on the total for the next control group.

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.

Chapter 18. 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 may be defined in the main source section of the module containing the subprocedure definition. If the prototype is not specified, the prototype is implicitly defined using the information in the procedure interface. If the procedure interface is also not defined, a default prototype with no return value and no parameters is implicitly defined.

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, if the prototype is specified.
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 an example of a subprocedure, see Figure 5 on page 22.

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 166. 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 167. 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 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 procedure-name line.
2. One main procedure-name line, containing a name, begin/end procedure, and keywords. If a continued name line is coded, the Name entry of the main procedure-name line may be left blank.
3. Zero or more keyword continuation lines.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 10
PContinuedName+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Comments++++++++++++
```

Figure 168. Procedure Specification Continued Name Line Layout

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 normal rules for RPG IV apply; reserved words cannot be used (see "Symbolic Names" on page 3). 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 a prototype is specified. If a prototype is not specified, the prototype will be implicitly defined using the name specified on the Procedure Specification and the information specified by the procedure interface.

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

EXPORT

The specification of the EXPORT keyword allows the procedure to be called by another module in the program. The name in positions 7-21 is exported in uppercase form.

Note: Procedure names are not imported using the IMPORT keyword. They are imported implicitly by any module in the program that makes a bound call to the procedure or that uses the procedure name to initialize a procedure pointer.

If the EXPORT keyword is not specified, the procedure can only be called from within the module.

SERIALIZE

When the SERIALIZE keyword is specified in a concurrent-thread module, only one thread can run in the procedure at any time. If one thread is running in the procedure and another thread calls the procedure, the second thread will wait to run the procedure until the first thread is no longer running in the procedure. If a thread is running in the procedure and it makes a recursive call to the procedure, then it must return from all the recursive calls to the procedure before another thread can begin running in the procedure.

The SERIALIZE keyword is allowed only when THREAD(*CONCURRENT) is specified on the Control specification.

Specifying SERIALIZE for one procedure is similar to specifying THREAD(*SERIALIZE) on the control specification. The difference is that specifying THREAD(*SERIALIZE) on the Control specification limits access by multiple threads to all the procedures in the module, while specifying the SERIALIZE keyword for a procedure only limits access to that procedure.

If you have more than one procedure in a module with the SERIALIZE keyword, the procedures are independent. One thread can be running in one serialized procedure, while another thread is running in another serialized procedure in the same module. For example, if procedures PROCA and PROCB in the same module both have the SERIALIZE keyword, one thread could be running PROCA while another thread was running PROCB. For more information on using serialized procedures, see “THREAD(*CONCURRENT | *SERIALIZE)” on page 275.

Part 4. Operations, Expressions, and Functions

This section describes the various ways in which you can manipulate data or devices. The major topics include:

- Operations that you can perform, using operation codes or built-in functions
- Expressions and the rules governing them
- Built-in functions
- Operation codes.

Chapter 19. Operations

The RPG IV 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 22, “Operation Codes,” on page 607 or Chapter 21, “Built-in Functions,” on page 493.

Operation Codes

The following table shows the free-form syntax for each operation code.

- Extenders
  - **(A)** Always perform a dump, even if DEBUG(*NO) is specified
  - **(A)** Sort ascending
  - **(D)** Pass operational descriptors on bound call
  - **(D)** Date field
  - **(D)** Sort descending
  - **(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 54. Operation Codes in Free-Form Syntax

Code	Free-Form Syntax
ACQ[1]	ACQ{(E)} *device-name workstn-file*
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*
COMMIT	COMMIT{(E)} {*boundary*}
DEALLOC [1]	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-queue* {*response*}}}

Table 54. Operation Codes in Free-Form Syntax (continued)

Code	Free-Form Syntax
DUMP[1]	DUMP{(A)} {*identifier*}
ELSE	ELSE
ELSEIF	ELSEIF{(MR)} *indicator-expression*
ENDDO	ENDDO
ENDFOR	ENDFOR
ENDIF	ENDIF
ENDMON	ENDMON
ENDSL	ENDSL
ENDSR	ENDSR {*return-point*}
EVAL	{EVAL{(HMR)}} *result = expression*
EVALR	EVALR{(MR)} *result = expression*
EVAL-CORR	EVAL-CORR{(EH)} *target-ds = source-ds*
EXCEPT	EXCEPT {*except-name*}
EXFMT	EXFMT{(E)} *format-name {data-structure}*
EXSR	EXSR *subroutine-name*
FEOD	FEOD{(EN)} *file-name*
FOR	FOR{(MR)} *index* {= *start*} {BY *increment*} {TO | DOWNTO *limit*}
FORCE	FORCE *file-name*
IF	IF{(MR)} *indicator-expression*
IN [1]	IN{(E)} {*LOCK} *data-area-name*
ITER	ITER
LEAVE	LEAVE
LEAVESR	LEAVESR
MONITOR	MONITOR
NEXT[1]	NEXT{(E)} *program-device file-name*
ON-ERROR	ON-ERROR {*exception-id1* {*:exception-id2...*}}
OPEN	OPEN{(E)} *file-name*
OTHER	OTHER
OUT[1]	OUT{(E)} {*LOCK} *data-area-name*
POST [1]	POST{(E)} {*program-device*} *file-name*
READ	READ{(EN)} *file-or-record-name* {*data-structure*}
READC	READC{(E)} *record-name* {*data-structure*}
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*}
REL [1]	REL{(E)} *program-device file-name*
RESET [1]	RESET{(E)} {*NOKEY} {*ALL} *name*
RETURN	RETURN{(HMR)} *expression*
ROLBK	ROLBK{(E)}
SELECT	SELECT

Table 54. Operation Codes in Free-Form Syntax (continued)

Code	Free-Form Syntax
SETGT	SETGT{(EHMR)} *search-arg file-or-record-name*
SETLL	SETLL{(EHMR)} *search-arg file-or-record-name*
SORTA	SORTA{(AD)} *array-name* or *keyed-ds-array*
TEST [1]	TEST{(EDTZ)} {*dtz-format*} *field-name*
UNLOCK [1]	UNLOCK{(E)} *name*
UPDATE	UPDATE{(E)} *file-or-record-name* {*data-structure* | %FIELDS(*name*{*:name...*})}
WHEN	WHEN{(MR)} *indicator-expression*
WRITE	WRITE{(E)} *file-or-record-name* {*data-structure*}
XML-INTO	XML-INTO{(EH)} *target-or-handler xml-document*
XML-SAX	XML-SAX{(E)} *handler xml-document*

Notes:

1. Complex-qualified names are note allowed for this operation code.

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
  - **(A)** Always perform a dump, even if DEBUG(*NO) is specified
  - **(A)** Sort ascending
  - **(D)** Pass operational descriptors on bound call
  - **(D)** Date field
  - **(D)** Sort descending
  - **(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
  - **(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 55. Operation Codes in Traditional Syntax

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
ACQ (E[7])	device-name	workstn-file			ER	
ADD (H)	Addend	Addend	Sum	+	–	Z
ADDDUR (E)	Date/Time	Duration:Duration Code	Date/Time		ER	
ALLOC (E)		Length	Pointer		ER	
ANDxx	Comparand	Comparand				
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	LR
CALLB (D E)		Procedure name or Procedure pointer	Plist name		ER	LR
CALLP (E 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 or record format)			
CLOSE (E)		file-name or *ALL			ER	
COMMIT (E)	boundary				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	name (file or record format)		NR[2]	ER	
DIV (H)	Dividend	Divisor	Quotient	+	–	Z

Table 55. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
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)[4]	message	message-queue	response		ER	
DUMP (A)	identifier					
ELSE						
ELSEIF (M/R)		indicator-expression				
END		Increment value				
ENDCS						
ENDDO		Increment value				
ENDFOR						
ENDIF						
ENDMON						
ENDSL						
ENDSR	label	return-point				
EVAL (H M/R)		Result = Expression				
EVALR (M/R)		Result = Expression				
EVAL-CORR		EVAL-CORR *target-ds* = *source-ds*				
EXCEPT		except-name				
# **EXFMT (E)**		Record format-name	data-structure		ER	
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				
FORCE		file-name				
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[6]
LOOKUP[1] **(table)**	Search argument	Table name	Table name	HI	LO	EQ[6]
MHHZO		Source field	Target field			

Table 55. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
MHLZO		Source field	Target field			
MLHZO		Source field	Target field			
MLLZO		Source field	Target field			
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
NEXT (E)	program-device	file-name			ER	
OCCUR (E)	Occurrence value	Data structure	Occurrence value		ER	
ON-ERROR		Status codes				
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]	program-device	file-name	INFDS name		ER	
READ (E N)		name (file or record format)	data-structure		ER	EOF[5]
READC (E)		record-name	data-structure		ER	EOF[5]
READE (E N)	search-arg	name (file or record format)	data-structure		ER	EOF[5]
READP (E N)		name (file or record format)	data-structure		ER	BOF[5]
READPE (E N)	search-arg	name (file or record format)	data-structure		ER	BOF[5]
REALLOC (E)		Length	Pointer		ER	
REL (E)	program-device	file-name			ER	
RESET (E)	*NOKEY	*ALL	name (variable or record format)		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						
SETGT (E)	search-arg	name (file or record format)		NR[2]	ER	

Table 55. Operation Codes in Traditional Syntax (continued)

Codes	Factor 1	Factor 2	Result Field	Resulting Indicators		
				71-72	73-74	75-76
SETLL (E)	search-arg	name (file or record format)		NR[2]	ER	EQ[6]
SETOFF[1]				OF	OF	OF
SETON[1]				ON	ON	ON
SHTDN				ON		
SORTA (A/D)		array-name or keyed-ds-array				
SQRT (H)		Value	Root			
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)[8]			Date/Time or Timestamp Field		ER	
TEST (D E)[8]	Date Format		Character or Numeric field		ER	
TEST (E T)[8]	Time Format		Character or Numeric field		ER	
TEST (E Z)[8]	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	AI	JR	XX
TIME			Target field			
UNLOCK (E)		name (file or data area)			ER	
UPDATE (E)		name (file or record format)	data-structure		ER	
WHEN (M/R)		indicator-expression				
WHENxx	Comparand	Comparand				
WRITE (E)		name (file or record format)	data-structure		ER	EOF[5]
XFOOT (H)		Array name	Sum	+	–	Z
XLATE (E P)	From:To	String:start	Target String		ER	
XML-INTO		XML-INTO *target-or-handler xml-document*				
XML-SAX		XML-SAX{(E)} *handler xml-document*				
Z-ADD (H)		Addend	Sum	+	–	Z
Z-SUB (H)		Subtrahend	Difference	+	–	Z

Table 55. 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. You must specify factor 1 or the result field. You may specify both.
5. The %EOF built-in function can be used as an alternative to specifying an EOF or BOF resulting indicator.
6. The %EQUAL built-in function can be used to test the SETLL and LOOKUP operations.
7. For all operation codes with extender 'E', either the extender 'E' or an ER error indicator can be specified, but not both.
8. You must specify the extender 'E' or an error indicator for the TEST operation.

Built-in Functions

Built-in functions are similar to operation codes in that 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.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++
 *
 * This example shows a complex expression with multiple
 * nested built-in functions.
 *
 * %TRIM takes as its argument a string.  In this example, the
 * argument is the concatenation of string A and the string
 * returned by the %SUBST built-in function.  %SUBST will return
 * a substring of string B starting at position 11 and continuing
 * for the length returned by %SIZE minus 20.  %SIZE will return
 * the length of string B.
 *
 * If A is the string '     Toronto,' and B is the string
 * '  Ontario, Canada            ' then the argument for %TRIM will
 * be '     Toronto, Canada     ' and RES will have the value
 * 'Toronto, Canada'.
 *
C                   EVAL      RES = %TRIM(A + %SUBST(B:11:%SIZE(B) - 20))
```

Figure 169. Built-in Function Arguments Example

See the individual built-in function descriptions for details on what arguments 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.

```
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * In the following example, CITY contains the string
 * 'Toronto, Ontario'.  The SCAN operation is used to locate the
 * separating blank, position 9 in this illustration.  SUBST
 * places the string 'Ontario' in field TCNTRE.
 *
 * Next, TCNTRE is compared to the literal 'Ontario' and
 * 1 is added to CITYCNT.
 *
C     ' '           SCAN      CITY          C
C                   ADD       1             C
C                   SUBST     CITY:C        TCNTRE
C    'Ontario'      IFEQ      TCNTRE
C                   ADD       1             CITYCNT
C                   ENDIF
 *
 * In this example, CITY contains the same value, but the
 * variable TCNTRE is not necessary since the %SUBST built-in
 * function returns the appropriate value.  In addition, the
 * intermediary step of adding 1 to C is simplified since
 * %SUBST accepts expressions as arguments.
 *
C     ' '           SCAN      CITY          C
C                   IF        %SUBST(CITY:C+1) = 'Ontario'
C                   EVAL      CITYCNT = CITYCNT+1
C                   ENDIF
```

Figure 170. 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+++++++++++++++++++++++++++
 *
 * In this example, CUSTNAME is a field in the
 * externally described data structure CUSTOMER.
 * If the length of CUSTNAME is changed, the attributes of
 * both TEMPNAME and NAMEARRAY would be changed merely by
 * recompiling.  The use of the %SIZE built-in function means
 * no changes to your code would be necessary.
 *
D CUSTOMER      E DS
D                 DS
D TEMPNAME                            LIKE(CUSTNAME)
D NAMEARRAY                      1    OVERLAY(TEMPNAME)
D                                     DIM(%SIZE(TEMPNAME))
```

Figure 171. 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.

Table 56. Built-In Functions

Name	Arguments	Value Returned
%ABS	numeric expression	absolute value of expression
%ADDR	variable name {: *DATA}	address of variable, or address of the data portion of a variable-length 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 {:format}	value in packed numeric 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	non-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
%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 format

Table 56. Built-In Functions (continued)

Name	Arguments	Value Returned
%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, SCAN)
		'0' otherwise
%GRAPH	character, graphic, or UCS-2 expression	value in graphic format
%HANDLER	handling procedure : communication area	not applicable
%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 field name	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
%PARMS	none	number of parameters passed to procedure
%PARMNUM	procedure-interface parameter name	number of a procedure-interface parameter
%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
%SCANRPL	scan string: replacement string: source string {:scan start position {:scan length}}	string produced by replacing scan string by replacement string in source string, with the scan starting at start position for the specified length
%SECONDS	number of seconds	number of seconds as a duration
%SHTDN		'1' if the system operator has requested shutdown
		'0' otherwise
%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 argument up 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

Table 56. Built-In Functions (continued)

Name	Arguments	Value Returned
%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, graphic, or UCS-2 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
%XML	xml document { : options }	not applicable
%YEARS	number of years	number of years as a duration

Arithmetic Operations

The arithmetic operations are shown in the following table.

Table 57. Arithmetic Operations

Operation	Traditional Syntax	Free-Form Syntax
Absolute Value	"%ABS (Absolute Value of Expression)" on page 493	
Add	"ADD (Add)" on page 609	+ operator
Divide	"DIV (Divide)" on page 657	/ operator or "%DIV (Return Integer Portion of Quotient)" on page 521
Division Remainder	"MVR (Move Remainder)" on page 752	"%REM (Return Integer Remainder)" on page 567
Multiply	"MULT (Multiply)" on page 751	* operator
Square Root	"SQRT (Square Root)" on page 820	"%SQRT (Square Root of Expression)" on page 578
Subtract	"SUB (Subtract)" on page 821	- operator
Zero and Add	"Z-ADD (Zero and Add)" on page 902	(not allowed)
Zero and Subtract	"Z-SUB (Zero and Subtract)" on page 903	(not allowed)

For examples of arithmetic operations, see Figure 172 on page 437.

Remember the following when specifying arithmetic operations:

- Arithmetic operations can be done only on numerics (including numeric subfields, numeric arrays, numeric array elements, numeric table elements, numeric named constants, numeric figurative constants, and 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 result of an arithmetic operation replaces the data that was in the result field.
- An arithmetic operation does not change factor 1 and factor 2 unless they are the same as the result field.
- 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 preceding DIV was not executed, then undesirable results may occur.
- For information on using arrays with arithmetic operations, see “Specifying an Array in Calculations” on page 171.

Ensuring Accuracy

- The length of any field specified in an arithmetic operation cannot exceed 63 digits. If the result exceeds 63 digits, digits are dropped from either or both ends, depending on the location of the decimal point.
- The TRUNCNBR option (as a command parameter or as a keyword on a control specification) determines whether truncation on the left occurs with numeric overflow or a runtime error is generated. Note that TRUNCNBR does not apply to calculations performed within expressions. If any overflow occurs within expressions calculations, a run-time message is issued. In addition, TRUNCNBR does not apply to arithmetic operations performed in integer or unsigned format.
- 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. Half adjust is not allowed if the result field is float.

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:
- All integer and unsigned operations are performed in 8-byte form.
- 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 172. Arithmetic Operations in Free-form Calculations

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments
C*
C*   In the following example, the initial field values are:
C*
C*                         A = 1
C*                         B = 10.0
C*                         C = 32
C*                         D = -20
C*                         E = 6
C*                         F = 10.0
C*                         G = 2.77
C*                         H = 70
C*                         J = .6
C*                         K = 25
C*                         L = 1.0, 1.7, -1.1                           Result:
C*
C                   ADD       1             A                 3 0         A = 002
C     B             ADD       C             V                 5 2         V = 042.00
C     B             ADD       D             V                             V = -10.00
C                   Z-ADD     C             V                             V = 032.00
C                   SUB       1             E                 3 0         E = 005
C     C             SUB       B             W                 5 1         W = 0022.0
C     C             SUB       D             W                             W = 0052.0
C                   Z-SUB     C             W                             W = -0032.0
C                   MULT      E             F                 3 0         F = 060
C     B             MULT      G             X                 8 4         X = 0027.7000
C     B             MULT      D             X                             X = -0200.0000
C                   DIV       B             H                 3 0         H = 007
C     C             DIV       J             Y                 6 2         Y = 0053.33
C                   MVR                     Z                 5 3         Z = 00.002
C                   SQRT      K             Z                             Z = 05.000
C                   XFOOT     L             Z                             Z = 01.600
```

Figure 173. Arithmetic Operations in Fixed-form Calculations

Array Operations

The array operations are shown in the following table.

Table 58. Array Operations

Operation	Traditional Syntax	Free-Form Syntax
Look Up Elements	"LOOKUP (Look Up a Table or Array Element)" on page 711	"%LOOKUPxx (Look Up an Array Element)" on page 551 or "%TLOOKUPxx (Look Up a Table Element)" on page 593
Number of Elements	"%ELEM (Get Number of Elements)" on page 527	
Move an Array	"MOVEA (Move Array)" on page 734	(not allowed)
Sort an Array	"SORTA (Sort an Array)" on page 815	
Subset an Array	"%SUBARR (Set/Get Portion of an Array)" on page 584	
Sum the Elements of an Array	"XFOOT (Summing the Elements of an Array)" on page 849	"%XFOOT (Sum Array Expression Elements)" on page 602

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 498
- "%BITNOT (Invert Bits)" on page 499
- "%BITOR (Bitwise OR Operation)" on page 500
- "%BITXOR (Bitwise Exclusive-OR Operation)" on page 501
- "BITOFF (Set Bits Off)" on page 615
- "BITON (Set Bits On)" on page 617
- "TESTB (Test Bit)" on page 831.

Table 59. 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 195 on page 502)

The BITOFF and BITON operations allow you to turn off and on specific bits in a field specified in the result field. The result field must be a one-position character field.

The TESTB operation compares the bits identified in factor 2 with the corresponding bits in the field named as the result field.

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 60. Branching Operations

Operation	**Traditional Syntax**	**Free-Form Syntax**
Compare and Branch	"CABxx (Compare and Branch)" on page 619	(not allowed)

Table 60. Branching Operations (continued)

Operation	Traditional Syntax	Free-Form Syntax
Go To	"GOTO (Go To)" on page 696	(not allowed)
Iterate	"ITER (Iterate)" on page 703	
Leave	"LEAVE (Leave a Do/For Group)" on page 708	
Leave a subroutine	"LEAVESR (Leave a Subroutine)" on page 710	
Tag	"TAG (Tag)" on page 828	(not allowed)

The GOTO operation (when used with a TAG operation) allows branching. When a GOTO operation occurs, the program branches to the specified label. The label can be specified before or after the GOTO operation. The label is specified by the TAG or ENDSR operation.

The TAG operation names the label that identifies the destination of a GOTO or CABxx operation.

The ITER operation transfers control from within a DO-group to the ENDDO statement of the DO-group.

The LEAVE operation is similar to the ITER operation; however, LEAVE transfers control to the statement ***following*** the ENDDO operation.

The LEAVESR operation causes control to pass to the ENDSR operation of a subroutine.

See each operation for an explanation of its function.

Call Operations

The call operations are shown in the following table.

Table 61. Call Operations

Operation	Traditional Syntax	Free-Form Syntax
Call Program or Procedure	• "CALL (Call a Program)" on page 621 • "CALLB (Call a Bound Procedure)" on page 622 • "CALLP (Call a Prototyped Procedure or Program)" on page 623	"CALLP (Call a Prototyped Procedure or Program)" on page 623
Identify Parameters	• "PARM (Identify Parameters)" on page 765 • "PLIST (Identify a Parameter List)" on page 768	PI or PR definition specification
Number of Parameters	"%PARMS (Return Number of Parameters)" on page 563	
Number of a Parameter	"%PARMNUM (Return Parameter Number)" on page 565	
Return	"RETURN (Return to Caller)" on page 795	

CALLP is one 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. The prototype may be explicitly defined using a Prototype definition, or it may be implicitly defined by the compiler from the Procedure Interface, if the procedure is defined in the same module as the call.

Call operations allow an RPG IV 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 prototyped 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 or service programs that are bound in the same program or service program
- Subprocedures in the same module

If the program or procedure is not defined in the same module as the call, 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.

If the procedure is defined in the same module as the call, it is not necessary to explicitly define a prototype. The prototype can be implicitly defined by the compiler using the information specified by the Procedure Interface for the procedure.

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 or procedures. Some advantages of using prototyped calls are:

- The syntax is simplified because no PARM or PLIST operations are required.
- For some parameters, you can pass literals and expressions.
- When calling procedures, you do not have to remember whether operational descriptors are required.
- The compiler helps you pass enough parameters, of the the correct type, format and length, by giving an error at compile time if the call is not correct.
- The compiler helps you pass parameters with the correct format and length for some types of parameters, by doing a conversion at run time.

Figure 174 on page 442 shows an example using the prototype ProcName, passing three parameters. The prototype ProcName could refer to either a program or a procedure. It is not important to know this when making the call; this is only important when 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 174. 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 the appropriate chapter in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*. For more information on defining prototypes and parameters, see "Prototypes and Parameters" on page 153.

Operational Descriptors

Sometimes it is necessary to pass a parameter to a procedure even though the data type is not precisely known to the called procedure, (for example, different types of strings). In these instances you can use operational descriptors to provide descriptive information to the called procedure regarding the form of the parameter. The additional information allows the procedure to properly interpret the string. You should only use operational descriptors when they are expected by the called procedure.

You can request operational descriptors for both prototyped and non-prototyped parameters. For prototyped calls, you specify the keyword OPDESC on the prototype definition. For non-prototyped parameters, you specify (D) as the operation code extender of the CALLB operation. In either case, operational descriptors are then built by the calling procedure and passed as hidden parameters to the called procedure.

When you have specified the OPDESC keyword for your own procedure, you can call APIs to find out information about the length and type of some of the parameters. These APIs require you to pass a parameter number to identify which parameter you are interested in. Usually, the number of a parameter can be obtained by simply counting the parameters in the prototype or procedure interface. However, when the RTNPARM keyword is specified, the number of each parameter is one higher than its apparent number. Use the %PARMNUM built-in function to get the number of a particular parameter instead of using a numeric literal. For more information, see "OPDESC" on page 348, "RTNPARM" on page 363 and "%PARMNUM (Return Parameter Number)" on page 565.

Parsing Program Names on a Call

Program names are specified in factor 2 of a CALL operation or as the parameter of the EXTPGM keyword on a prototype or procedure interface. 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 the non-blank data in a field or named constant, including the slash, cannot 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 re-resolved, and they do not affect whether or not a program reference by variable is re-resolved. Figure 175 on page 444 illustrates the grouping of program references.

Program CALL Example

```
*...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'
 *
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C                   CALL      Pgm_Ex_A
 *
 * The following two calls will be grouped together because both
 * have the same program name (PGM1) and the same library name
 * (none). Note that these will not be grouped with the call using
 * Pgm_Ex_A above because Pgm_Ex_A has a different library
 * name specified (LIB1).
 *
C                   CALL      'PGM1'
C                   CALL      Pgm_Ex_B
 *
 * The following two program references will be grouped together
 * because both have the same program name (PGM2) and the same
 * library name (LIB).
 *
C                   CALL      'LIB/PGM2'
C                   CALL      Pgm_Ex_C
 *
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The first call in the program using CALLV below will result in
 * a resolve being done for the variable CALLV to the program PGM1.
 * This is independent of any calls by a literal or named constant
 * to PGM1 that may have already been done in the program.  The
 * second call using CALLV will not result in a resolve to PGM1
 * because the value of CALLV has not changed.
 *
C                   MOVE     'PGM1'        CALLV         21
C                   CALL     CALLV
C                   CALL     CALLV
```

Figure 175. Example of Grouping of Program References

Parsing System Built-In Names

When the literal or named constant specified on a bound call starts with "CEE" or an underscore ('_'), the compiler will treat this as a system built-in. (A bound call results with either CALLB or with a prototyped call where EXTPGM is *not* specfied on the prototype).

If it is not actually a system built-in, then a warning will appear in the listing; you can ignore this warning.

For more information on APIs, see the iSeries Information Center programming category. To avoid confusion with system provided APIs, you should not name your procedures starting with "CEE".

Value of *ROUTINE

When a call fails, the contents of the *ROUTINE subfield of the program status data structure (PSDS) is updated with the following:

- On an external call, the name of the called program (that is, for CALL or CALLP to a program).
- On a bound static call, the name of the called procedure.
- On a bound procedure pointer call, *N.

Note that since the size of this subfield is only 8 bytes long, the name may be truncated.

Compare Operations

The compare operations are shown in the following table.

Table 62. Compare Operations

Operation	**Traditional Syntax**	**Free-Form Syntax**
And	"ANDxx (And)" on page 613	AND operator
Compare	"COMP (Compare)" on page 648	=, <, >, <=, >=, or <> operator
Compare and Branch	"CABxx (Compare and Branch)" on page 619	(not allowed)
Conditional Subroutine	"CASxx (Conditionally Invoke Subroutine)" on page 628	"IF (If)" on page 698 and "EXSR (Invoke Subroutine)" on page 688
Do Until	"DOU (Do Until)" on page 660 or "DOUxx (Do Until)" on page 661	"DOU (Do Until)" on page 660
Do While	"DOW (Do While)" on page 663 or "DOWxx (Do While)" on page 664	"DOW (Do While)" on page 663
If	"IF (If)" on page 698 or "IFxx (If)" on page 699	"IF (If)" on page 698
Or	"ORxx (Or)" on page 761	OR operator
When	"WHEN (When True Then Select)" on page 843 or "WHENxx (When True Then Select)" on page 844	"WHEN (When True Then Select)" on page 843

In 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 postions 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.
- Blanks within zoned numeric fields are assumed to be zeros, if the FIXNBR(*ZONED) control specification keyword or command parameter is used in the compilation of the program.
- 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.
- 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. See Figure 52 on page 133 for examples.

Attention!

Note the following points, especially if you want to avoid unpredictable results.

- The order of the characters is not necessarily the same for UCS-2 data as it is for character or graphic data; for example '2' is less than 'A' in UCS-2, but it is greater than 'A' for a character comparison. If a comparison operation involves implicit conversion to UCS-2, or if you change some of your fields to have UCS-2 type instead of character or graphic type, then you may notice that some less-than or greater-than comparisons have different results than you expect.
- All graphic and UCS-2 comparisons are done using the hexadecimal representation of the data. The alternate sequence is not used.
- If an alternate collating sequence (using the "ALTSEQ{(*NONE | *SRC | *EXT)}" on page 258 keyword on the Control specification) has been specified for the comparison of character fields, the comparands are converted to the alternate sequence and then compared. If *HIVAL or *LOVAL is used in the comparison, the alternate collating sequence may alter the value before the compare operation. Note that if either comparand is defined with the ALTSEQ(*NONE) keyword on the definition specification, the alternate collating sequence is not used.
- When comparing a basing pointer to *NULL (or to a basing pointer with value *NULL), the only comparisons that produce predictable results are for equality and inequality.
- Comparing pointers for less-than or greater-than produces predictable results only when the pointers point to addresses in contiguous storage. For example, all pointers are set to addresses in one *USRSPC, or all pointers are set to the addresses of array elements in one array.
- When procedure pointer fields are compared for anything except equality or inequality, the results will be unpredictable.
- Because of the way float values are stored, they should not be compared for equality or inequality. Instead, the absolute value of the difference between the two values should be compared with a very small value.

Conversion Operations

The following built-in functions perform conversion operations:

- "%CHAR (Convert to Character Data)" on page 505
- "%DEC (Convert to Packed Decimal Format)" on page 513
- "%DECH (Convert to Packed Decimal Format with Half Adjust)" on page 515
- "%EDITC (Edit Value Using an Editcode)" on page 522
- "%EDITFLT (Convert to Float External Representation)" on page 525
- "%EDITW (Edit Value Using an Editword)" on page 526
- "%FLOAT (Convert to Floating Format)" on page 534
- "%GRAPH (Convert to Graphic Value)" on page 537
- "%INT (Convert to Integer Format)" on page 544
- "%INTH (Convert to Integer Format with Half Adjust)" on page 544
- "%UCS2 (Convert to UCS-2 Value)" on page 599
- "%UNS (Convert to Unsigned Format)" on page 600

- "%UNSH (Convert to Unsigned Format with Half Adjust)" on page 600

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 720
- "MOVEL (Move Left)" on page 741

Data-Area Operations

The data-area operations are:
- "IN (Retrieve a Data Area)" on page 701
- "OUT (Write a Data Area)" on page 764
- "UNLOCK (Unlock a Data Area or Release a Record)" on page 839.

These operations are available in both the traditional syntax and free-form syntax.

The IN and OUT operations allow you to retrieve and write one or all data areas in a program, depending on the factor 2 entry.

The IN and OUT operations also allow you to control the locking or unlocking of a data area. When a data area is locked, it can be read but not updated by other programs or procedures.

The following lock states are used:
- For an IN operation with *LOCK specified, an exclusive allow read lock state is placed on the data area.
- For an OUT operation with *LOCK the data area remains locked after the write operation
- For an OUT operation with blank the data area is unlocked after it is updated
- UNLOCK is used to unlock data areas and release record locks, the data areas and/or records are not updated.

During the actual transfer of data into or out of a data area, there is a system-internal lock on 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.

Remember the following when using the IN, OUT, and UNLOCK operations:
- A data-area operation cannot be done on a data area that is not defined to the operating system.
- Before the IN, OUT, and UNLOCK operations can be done on a data area, you must specify the DTAARA keyword on the definition specification for the data area, or specify the data area in the result field of an *DTAARA DEFINE statement. (For further information on the DEFINE statement, see "DEFINE (Field Definition)" on page 651.)
- A locked data area cannot be updated or locked by another RPG program; however, the data area can be retrieved by an IN operation with factor 1 blank.
- 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.

A data structure defined with a U in position 23 of the definition specifications indicates that the data structure is a data area. You may specify the DTAARA keyword for a data area data structure, if specified you can use the IN, OUT and UNLOCK operation codes to specify further operations for the data area. The data area is automatically read and locked at program initialization time, and the contents of the data structure are written to the data area when the program ends with LR on. If the data area for a data area data structure is not found, it will be created with an initial value of blanks. If the library list was searched for the data area, the new data area will be created in QTEMP.

To define the local data area (*LDA) you can do one of the following:
- Specify the DTAARA(*LDA) keyword on the definition specification for the data area.
- Specify UDS on the definition specification for the data area and leave the name blank.
- Specify *LDA in factor 2 of a *DTAARA DEFINE statement.

To define the *PDA you may specify the DTAARA(*PDA) keyword on the definition specification for the data area, or specify *PDA in factor 2 of a *DTAARA DEFINE statement.

Date Operations

The date operations are shown in the following table.

Table 63. Date Operations

Operation	Traditional Syntax	Free-Form Syntax
Add Duration	“ADDDUR (Add Duration)” on page 610	+ operator
Extract	“EXTRCT (Extract Date/Time/Timestamp)” on page 689	“%SUBDT (Extract a Portion of a Date, Time, or Timestamp)” on page 587
Subtract Duration	“SUBDUR (Subtract Duration)” on page 822	- operator or “%DIFF (Difference Between Two Date, Time, or Timestamp Values)” on page 518
Convert date/time/timestamp to character	“MOVE (Move)” on page 720 or “MOVEL (Move Left)” on page 741	“%CHAR (Convert to Character Data)” on page 505
Convert date/time/timestamp to numeric	“MOVE (Move)” on page 720 or “MOVEL (Move Left)” on page 741	“%DEC (Convert to Packed Decimal Format)” on page 513

Table 63. Date Operations (continued)

Operation	Traditional Syntax	Free-Form Syntax
Convert character/numeric to date	"MOVE (Move)" on page 720 or "MOVEL (Move Left)" on page 741	"%DATE (Convert to Date)" on page 511
Convert character/numeric to time	"MOVE (Move)" on page 720 or "MOVEL (Move Left)" on page 741	"%TIME (Convert to Time)" on page 591
Convert character/numeric/date to timestamp	"MOVE (Move)" on page 720 or "MOVEL (Move Left)" on page 741	"%TIMESTAMP (Convert to Timestamp)" on page 592
Move date/time to timestamp	"MOVE (Move)" on page 720 or "MOVEL (Move Left)" on page 741	date + time
Test	"TEST (Test Date/Time/Timestamp)" on page 829	
Number of Years	"%YEARS (Number of Years)" on page 606	
Number of Months	"%MONTHS (Number of Months)" on page 555	
Number of Days	"%DAYS (Number of Days)" on page 512	
Number of Hours	"%HOURS (Number of Hours)" on page 543	
Number of Minutes	"%MINUTES (Number of Minutes)" on page 554	
Number of Seconds	"%SECONDS (Number of Seconds)" on page 574	
Number of Microseconds	"%MSECONDS (Number of Microseconds)" on page 556	

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 64. 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 64 on page 450.

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 65. 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 66. Declarative Operations

Operation	Traditional Syntax	Free-Form Syntax
Define Field	"DEFINE (Field Definition)" on page 651	LIKE or DTAARA keyword on definition specification
Define Key	• "KFLD (Define Parts of a Key)" on page 705 • "KLIST (Define a Composite Key)" on page 706	(not allowed)
Identify Parameters	• "PARM (Identify Parameters)" on page 765 • "PLIST (Identify a Parameter List)" on page 768	PR definition specification
Tag	"TAG (Tag)" on page 828	(not allowed)

The declarative operations do not cause an action to occur (except PARM with optional factor 1 or 2); they can be specified anywhere within calculations. They 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.

The DEFINE operation either defines a field based on the attributes (length and decimal positions) of another field or defines a field as a data area.

The KLIST and KFLD operations are used to indicate the name by which a composite key field may be referred and the fields that compose the composite key. A *composite key* is a key that contains a list of key fields. It is built from left to right, with the first KFLD specified being the leftmost (high-order) field of the composite key.

The PLIST and PARM operations are used with the CALL and CALLB operations to allow a called program or procedure access to parameters from a calling program or procedure.

The TAG operation names the destination of a branching operation such as GOTO or CABxx.

Error-Handling Operations

The exception-handling operation codes are:
- "MONITOR (Begin a Monitor Group)" on page 718
- "ON-ERROR (On Error)" on page 758
- ENDMON, as described in "ENDyy (End a Structured Group)" on page 673

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 176. Example of MONITOR and ON-ERROR blocks

File Operations

The file operation codes are:
- "ACQ (Acquire)" on page 608
- "CHAIN (Random Retrieval from a File)" on page 633
- "CLOSE (Close Files)" on page 646
- "COMMIT (Commit)" on page 647
- "DELETE (Delete Record)" on page 655
- "EXCEPT (Calculation Time Output)" on page 684
- "EXFMT (Write/Then Read Format)" on page 686
- "FEOD (Force End of Data)" on page 691
- "FORCE (Force a Certain File to Be Read Next Cycle)" on page 695
- "NEXT (Next)" on page 753

The file built-in functions are:

These operations are available in both the traditional syntax and free-form syntax.

Most file operations can be used with both program described and externally described files (F or E respectively in position 22 of the file description specifications).

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.

When the OVRDBF (override with data base file) command is used with the MBR (*ALL) parameter specified, the SETLL, SETGT and CHAIN operations only process the current open file member. For more information, refer to the see the iSeries Information Center database and file systems category.

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, EXFMT, 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.

Exception/errors that occur during file operations can be handled by the programmer (by coding an error indicator or specifying a file-error subroutine), or by the RPG IV 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 219 for information on handling files with null-capable fields.

You can pass a file as a parameter to a prototyped program or procedure. When you pass a file as a parameter, then any settings for the file that are defined using File specification keywords are in effect for all procedures that access the file. For example, if the EXTFILE keyword is specified with a variable parameter, and a called procedure opens the file, then the value of the caller's variable will be used to set the name of the file to be opened. If the called procedure needs to change or access those variables associated with the file through keywords, the calling procedure must pass the variables as a parameter.

The file-feedback built-in functions %EOF(filename), %EQUAL(filename), %FOUND(filename), %OPEN(filename), and %STATUS(filename) can be used in the called procedure program or to determine the current state of the file by specifying the name of the file parameter as the operand to the built-in function.

For more information on file parameters, see “LIKEFILE(filename)” on page 343 and Chapter 6, “General File Considerations,” on page 107.

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 812
- “SETON (Set Indicator On)” on page 813

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 557

The SETON and SETOFF operations set (on or off) indicators specified in positions 71 through 76. At least one resulting indicator must be specified in these positions. Remember the following when setting indicators:

- The 1P, MR, KA through KN, and KP through KY indicators cannot be set on by the SETON operation.
- The 1P and MR indicators cannot be set off by the SETOFF operation.
- Setting L1 through L9 on or off with a SETON or SETOFF operation does not set any lower control level indicators.

Information Operations

The information operations are shown in the following table.

Table 67. Information Operations

Operation	Traditional Syntax	Free-Form Syntax
Dump	"DUMP (Program Dump)" on page 669	
Get Shutdown Status	"SHTDN (Shut Down)" on page 814	"%SHTDN (Shut Down)" on page 575
Get Time and Date	"TIME (Retrieve Time and Date)" on page 837	• "%DATE (Convert to Date)" on page 511 • "%TIME (Convert to Time)" on page 591 • "%TIMESTAMP (Convert to Timestamp)" on page 592

The DUMP operation provides a dump of all indicators, fields, data structures, arrays, and tables used in a program.

The SHTDN operation allows the program to determine whether the system operator has requested shutdown. If so, the resulting indicator that must be specified in positions 71 and 72 is set on.

The TIME operation allows the program to access the system time of day and system date at any time during program running.

Initialization Operations

The initialization operations provide run-time clearing and resetting of all elements in a structure (record format, data structure, array, or table) or a variable (field, subfield, or indicator).

The initialization operations are:
- "CLEAR (Clear)" on page 642
- "RESET (Reset)" on page 788.

These operations are available in both the traditional syntax and free-form syntax.

The CLEAR operation sets all elements in a structure or variable to their default value depending on the field type (numeric, character, graphic, UCS-2, indicator, pointer, or date/time/timestamp).

The RESET operation sets all elements in a structure or variable to their initial values (the values they had at the end of the initialization step in the program cycle).

The RESET operation is used with data structure initialization and the initialization subroutine (*INZSR). You can use both data structure initialization and the *INZSR to set the initial value of a variable. The initial value will be used to set the variable if it appears in the result field of a RESET operation.

When these operation codes are applied to record formats, only fields which are output are affected (if factor 2 is blank) or all fields (if factor 2 is *ALL). The factor 1 entry of *NOKEY prevents key fields from being cleared or reset.

*ALL may be specified in factor 2 if the result field contains a table name, or multiple occurrence data structure or record format. If *ALL is specified all elements or occurrences will be cleared or reset. See "CLEAR (Clear)" on page 642 and "RESET (Reset)" on page 788 for more detail.

For more information see Chapter 9, "Data Types and Data Formats," on page 179.

Memory Management Operations

The memory management operations are shown in the following table.

Table 68. Memory Management Operations

Operation	Traditional Syntax	Free-Form Syntax
Allocate Storage	"ALLOC (Allocate Storage)" on page 612	"%ALLOC (Allocate Storage)" on page 497
Free Storage	"DEALLOC (Free Storage)" on page 649	
Reallocate Storage	"REALLOC (Reallocate Storage with New Length)" on page 785	"%REALLOC (Reallocate Storage)" on page 566
Get the Address of a Variable	"%ADDR (Get Address of Variable)" on page 494	
Get the Address of a Procedure	"%PADDR (Get Procedure Address)" on page 560	

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.

There are two types of heap storage: single-level and teraspace. You can use the ALLOC keyword on the Control specification to control which type of heap storage is used by your memory management operations.

There are advantages and disadvantages of each type of heap storage.

- The maximum size of an individual allocation or reallocation is larger for teraspace heap storage.

  - The maximum size that RPG allows for the %ALLOC and %REALLOC built-in functions is 4294967295 bytes. When you use single-level heap storage, the maximum size that RPG allows is 16776704 bytes.
  - RPG allows the larger maximum of 4294967295 bytes for the ALLOC and REALLOC operation codes when the compiler can detect at compile time that memory management operations will use teraspace heap storage. If RPG memory management operations will use single-level heap storage, or if the compiler cannot detect the type of heap storage at compile time, then the smaller limit of 16776704 bytes will be in effect.
  - Note that the actual maximum size that you can allocate may be less than the maximum size that RPG allows, depending on the availability of heap storage at runtime.
- The system functions that RPG uses to reallocate and deallocate teraspace heap storage can handle pointers to either single-level heap storage or teraspace heap storage. When the teraspace reallocation function is used to reallocate a pointer, the new allocation will be the same type of heap storage as the original allocation.
- The system functions that RPG uses to reallocate and deallocate single-level heap storage can only handle pointers to single-level heap storage.
- Single-level storage can provide greater integrity than teraspace storage. For example, using single-level storage, the storage that can be affected by a storage over-run is measured in megabytes; for teraspace storage, it is measured in terabytes.

For more information on the different types of heap storage, see the chapter on storage management in *ILE Concepts*, SC41-5606-09.

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
 * After this point, Fld1 should not be accessed since the
 * basing pointer Ptr1 no longer points to allocated storage.
C                 CALL      'SOMEPGM'

 * During the previous call to 'SOMEPGM', several storage allocations
 * may have been done.  In any case, it is extremely dangerous to
 * make the following assignment, since 25 bytes of storage will
 * be filled with 'a'.  It is impossible to know what that storage
 * 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 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 Operation

The message operation

- "DSPLY (Display Message)" on page 666

allows interactive communication between the program and the operator or between the program and the display workstation that requested the program.

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 69. Move Operations

Operation	Traditional Syntax	Free-Form Syntax
Move	"MOVE (Move)" on page 720	"EVALR (Evaluate expression, right adjust)" on page 678 or conversion built-in functions
Move an Array	"MOVEA (Move Array)" on page 734	(not allowed)
Move Left	"MOVEL (Move Left)" on page 741	"EVAL (Evaluate expression)" on page 676 or conversionbuilt-in functions

Move operations transfer all or part of factor 2 to the result field. Factor 2 remains unchanged.

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. (See Figure 345 on page 732 for an example.) 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. (See Figure 347 on page 744 for an example.)

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'oXXi' (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 *ALL, the compiler will check to make sure it is entirely enclosed by one pair of shift-out shift-in characters (SO/SI). The compiler also checks that the character source is of even length and at least 4 bytes (SO/SI plus one graphic character). When moving from a hexadecimal literal or *ALLX to graphic field, the first byte and last byte of the hexadecimal literal or the pattern within *ALLX must not be 0E (shift out) and 0F (shift in). But the hexadecimal literal (or pattern) should still represent an even number of bytes.

When a character field is involved in a move from/to a graphic field, the compiler will check that the character field is of even length and at least 4 bytes long. At runtime, the compiler checks the content of the character field to make sure it is entirely enclosed by only one pair of SO/SI.

When moving from a graphic field to a character field, if the length of the character field is greater than the length of the graphic field (in bytes) plus 2 bytes, the SO/SI are added immediately before and after the graphic data. This may cause unbalanced SO/SI in the character field due to residual data in the character field, which will not be diagnosed by the compiler.

When move operations are used to move data from character fields to graphic fields, shift-out and shift-in characters are removed. When moving data from graphic fields to character fields, shift-out and shift-in characters are inserted in the target field.

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 shift characters and 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 with shift characters separating the modes

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, and contains matched SO/SI pairs.

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 for numeric, and '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
- Time to Time
- Timestamp to Timestamp
- Date to Timestamp
- Time to Timestamp (sets micro-seconds to 000000)
- Timestamp to Date
- Timestamp to Time
- Date to Character or Numeric
- Time to Character or Numeric
- 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 Type" on page 206, "Time Data Type" on page 208, and "Timestamp Data Type" on page 210.

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, and a special value *JOBRUN are allowed in factor 1. (For more information, see Table 35 on page 208.)
- A zero (0) specified at the end of a format (for example *MDY0) indicates that the character field does not contain separators.
- A 2-digit year format (*MDY, *DMY, *YMD, *JUL and *JOBRUN) 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 these ranges.
- 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 2-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 used 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 will be generated if it is not valid.

When moving from a Time to a Timestamp field, the microseconds part of the timestamp is 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.

Examples of Converting a Character Field to a Date Field

Figure 177 on page 465 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 177. Moving character and date data

The following example shows how to convert from a character field in the form CYYMMDD to a date field in *ISO format. This is particularly useful when using command parameters of type *DATE.

The RPG program is only intended to be called using the command interface, so it is not necessary to specify a prototype for the program. The prototype will be implicitly defined by the compiler using the information in the procedure interface.

```
              CMD        PROMPT('Use DATE parameter')
              PARM       KWD(DATE) TYPE(*DATE)
```

Figure 178. Source for a command using a date parameter.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 * Procedure interface for this program (no prototype is necessary)
D FIG210          PI                  EXTPGM('FIG210')
D   DateParm                     7A

 * Declare a date type with date format *ISO.
D ISO_DATE        S               D   DATFMT(*ISO)

 * The format of the DateParm parameter is CYYMMDD, so code
 * *CYMD0 as the 2nd parameter of built-in function %DATE.
 /FREE
    ISO_DATE = %DATE (DateParm: *CYMD0);
 /END-FREE
```

Figure 179. Part of RPG IV command processing program for this command.

Move Zone Operations

The move zone operations are:

- "MHHZO (Move High to High Zone)" on page 714
- "MHLZO (Move High to Low Zone)" on page 715
- "MLHZO (Move Low to High Zone)" on page 716
- "MLLZO (Move Low to Low Zone)" on page 717.

These operations are available only in the traditional syntax.

The move zone operations move only the zone portion of a character.

Whenever the word *high* is used in a move zone operation, the field involved must be a character field; whenever *low* is used, the field involved can be either a character or a numeric field. Float numeric fields are not allowed in the Move Zone operations.

Characters J through R have D zones and can be used to obtain a negative value: `(J = hexadecimal D1, ..., R = hexadecimal D9)`.

Note: While you may see this usage in old programs, your code will be clearer if you use hexadecimal literals for this purpose. Use X'F0' to obtain a positive zone and X'D0' to obtain a negative zone.

Note: The character (-) is represented by a hexadecimal 60, and cannot be used to obtain a negative result, since it has a zone of 6, and a negative result requires a zone of "D".

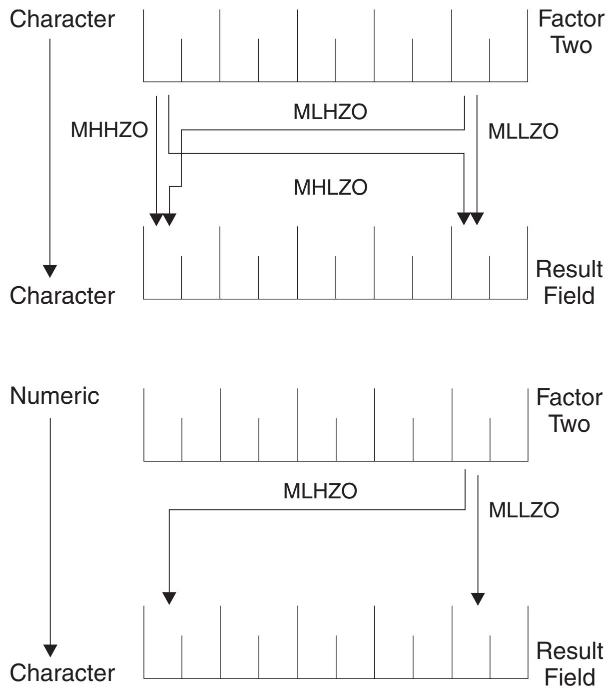


Figure 180. Function of MOVE Zone Operations

Result Operations

The following built-in functions work with the result of the previous operation:

- "%EQUAL (Return Exact Match Condition)" on page 530
- "%FOUND (Return Found Condition)" on page 535
- "%ERROR (Return Error Condition)" on page 532
- "%STATUS (Return File or Program Status)" on page 579

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 517
- "%LEN (Get or Set Length)" on page 547
- "%SIZE (Get Size in Bytes)" on page 576

These built-in functions are available in both the traditional syntax and free-form syntax.

String Operations

The string operations are shown in the following table.

Table 70. String Operations

Operation	Traditional Syntax	Free-Form Syntax
Concatenate	"CAT (Concatenate Two Strings)" on page 630	+ operator
Check	"CHECK (Check Characters)" on page 636	"%CHECK (Check Characters)" on page 507
Check Reverse	"CHECKR (Check Reverse)" on page 639	"%CHECKR (Check Reverse)" on page 509
Create	"%STR (Get or Store Null-Terminated String)" on page 582	
Replace	"%REPLACE (Replace Character String)" on page 568	
Scan	"SCAN (Scan String)" on page 799	"%SCAN (Scan for Characters)" on page 570
Scan and Replace	"%SCANRPL (Scan and Replace Characters)" on page 572	
Substring	"SUBST (Substring)" on page 825	"%SUBST (Get Substring)" on page 588
Translate	"XLATE (Translate)" on page 850	"%XLATE (Translate)" on page 603
Trim Blanks	"%TRIM (Trim Characters at Edges)" on page 595, "%TRIML (Trim Leading Characters)" on page 597, or "%TRIMR (Trim Trailing Characters)" on page 598	

The string operations include concatenation, scanning, substringing, translation, and verification. String operations can only be used on character, graphic, or UCS-2 fields.

The CAT operation concatenates two strings to form one.

The CHECK and CHECKR operations verify that each character in factor 2 is among the valid characters in factor 1. CHECK verifies from left to right and CHECKR from right to left.

The SCAN operation scans the base string in factor 2 for occurrences of another string specified in factor 1.

The SUBST operation extracts a specified string from a base string in factor 2. The extracted string is placed in the result field.

The XLATE operation translates characters in factor 2 according to the from and to strings in factor 1.

Note: 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.

In the string operations, factor 1 and factor 2 may have two parts. If both parts are specified, they must be separated by a colon. This option applies to all but the CAT, CHECK, CHECKR, and SUBST operations (where it applies only to factor 2).

If you specify P as the operation extender for the CAT, SUBST, or XLATE operations, the result field is padded from the right with blanks after the operation.

See each operation for a more detailed explanation.

When using string operations on graphic fields, all data in factor 1, factor 2, and the 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. Preserving data integrity is the user's responsibility.

Structured Programming Operations

The structured programming operations are shown in the following table.

Table 71. Structured Programming Operations

Operation	Traditional Syntax	Free-Form Syntax
And	"ANDxx (And)" on page 613	AND operator
Do	"DO (Do)" on page 658	"FOR (For)" on page 692
Do Until	"DOU (Do Until)" on page 660 or "DOUxx (Do Until)" on page 661	"DOU (Do Until)" on page 660
Do While	"DOW (Do While)" on page 663 or "DOWxx (Do While)" on page 664	"DOW (Do While)" on page 663
Else	"ELSE (Else)" on page 671	
Else If	"ELSEIF (Else If)" on page 672	
End	"ENDyy (End a Structured Group)" on page 673	
For	"FOR (For)" on page 692	
If	"IF (If)" on page 698 or "IFxx (If)" on page 699	"IF (If)" on page 698
Iterate	"ITER (Iterate)" on page 703	
Leave	"LEAVE (Leave a Do/For Group)" on page 708	
Or	"ORxx (Or)" on page 761	OR operator
Otherwise	"OTHER (Otherwise Select)" on page 762	
Select	"SELECT (Begin a Select Group)" on page 802	
When	"WHEN (When True Then Select)" on page 843 or "WHENxx (When True Then Select)" on page 844	"WHEN (When True Then Select)" on page 843

The DO operation allows the processing of a group of calculations zero or more times starting with the value in factor 1, incrementing each time by a value on the associated ENDDO operation until the limit specified in factor 2 is reached.

The DOU and DOUxx (Do Until) operations allow the processing of a group of calculations one or more times. The end of a Do-Until operation is indicated by an ENDDO operation.

The DOW and DOWxx (Do While) operations allow the processing of a group of calculations zero or more times. The end of a Do-While operation is indicated by an ENDDO operation.

The FOR operation allows the repetitive processing of a group of calculations. A starting value is assigned to the index name. Increment and limit values can be specified, as well. Starting, increment, and limit values can be free-form expressions. An ENDFOR operation indicates the end of the FOR group.

The LEAVE operation interrupts control flow prematurely and transfers control to the statement following the ENDDO or ENDFOR operation of an iterative structured group. The ITER operation causes the next loop iteration to occur immediately.

The IF and IFxx operations allow the processing of a group of calculations if a specified condition is satisfied. The ELSE operation allows you to specify a group of calculations to be processed if the condition is not satisfied. The ELSEIF operation is a combination of an ELSE operation and an IF operation. The end of an IF or IFxx group is indicated by ENDIF.

The SELECT, WHEN, WHENxx, and OTHER group of operations are used to conditionally process one of several alternative sequences of operations. The beginning of the select group is indicated by the SELECT operation. The WHEN and WHENxx operations are used to choose the operation sequence to process. The OTHER operation is used to indicate an operation sequence that is processed when none of the WHENxx conditions are fulfilled. The end of the select group is indicated by the ENDSL operation.

The ANDxx and ORxx operations are used with the DOUxx, DOWxx, WHENxx, and IFxx operations to specify a more complex condition. The ANDxx operation has higher precedence than the ORxx operation. Note, however, that the IF, DOU, DOW, and WHEN operations allow a more straightforward coding of complex expressions than their xx counterparts.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * In the following example, indicator 25 will be set on only if the
 * first two conditions are true or the third condition is true.
 *
 * As an expression, this would be written:
 * EVAL *IN25 = ((FIELDA > FIELDB) AND (FIELDA >= FIELDC)) OR (FIELDA < FIELDD)
 *
 *
C     FIELDA        IFGT      FIELDB
C     FIELDA        ANDGE     FIELDC
C     FIELDA        ORLT      FIELDD
C                   SETON                                        25
C                   ELSE
C                   SETOFF                                       25
C                   ENDIF
```

Figure 181. Example of AND/OR Precedence

A DO, DOUxx, DOWxx, FOR, IFxx, MONITOR, or SELECT operation (with or without ANDxx or ORxx operations), and an ENDyy operation, delimit a structured group. The ENDDO operation ends each DO, DOUxx, and DOWxx group or causes the structured group to be reprocessed until the specified ending conditions are met. The ENDFOR operation ends each FOR group. The SELECT must end with an ENDSL. An IFxx operation and an IFxx operation with an ELSE operation must end with an ENDIF operation.

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 445.

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:

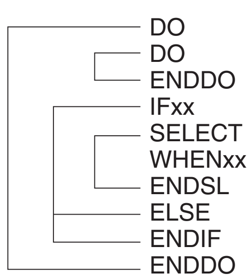


Figure 182. Nested Structured Groups

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.
- A structured group can be contained in detail, total, or subroutine calculations, but it cannot be split among them.
- Branching into a structured group from outside the structured group may cause undesirable results.

Subroutine Operations

The subroutine operations are:

- "BEGSR (Beginning of Subroutine)" on page 614
- "ENDSR (End of Subroutine)" on page 675
- "EXSR (Invoke Subroutine)" on page 688
- "LEAVESR (Leave a Subroutine)" on page 710
- "CASxx (Conditionally Invoke Subroutine)" on page 628 (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 procedure; 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.

Coding Subroutines

An RPG IV subroutine can be processed from any point in the calculation operations. All RPG IV operations can be processed within a subroutine, and these operations can be conditioned by any valid indicators in positions 9 through 11. SR or blanks can appear in positions 7 and 8. Control level indicators (L1 through L9) cannot be used in these positions. However, AND/OR lines within the subroutine can be indicated in positions 7 and 8.

Fields used in a subroutine can be defined either in the subroutine or in the rest of the procedure. In either instance, the fields can be used by both the body of the procedure and the subroutine.

A subroutine cannot contain another subroutine. One subroutine can call another subroutine; that is, a subroutine can contain an EXSR or CASxx. However, an EXSR or CASxx specification within a subroutine cannot directly call itself. Indirect calls to itself through another subroutine should not be performed, because unpredictable results will 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 statement.

The use of the GOTO (branching) 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.

A GOTO within a subroutine in the cycle-main procedure can be issued to a TAG within the same subroutine, detail calculations or total calculations. 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.

Subroutine Coding Examples

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * For a subroutine, positions 7 and 8 can be blank or contain SR.
 *
C                          :
C                          :
C                   EXSR      SUBRTB
C                          :
C                          :
C                          :
CL2                 EXSR      SUBRTA
C                          :
C                          :
C                          :
C     SUBRTA        BEGSR
C                          :
C                          :
C                          :
 *
 *  One subroutine can call another subroutine.
 *
C                   EXSR      SUBRTC
C                          :
C                          :
C                          :
C                   ENDSR
C     SUBRTB        BEGSR
C                          :
C                          :
C                          :
 *
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  GOTO and TAG operations can be used within a subroutine.
 *
C     START         TAG
C                          :
C                          :
C                          :
C   23              GOTO      END
C                          :
C                          :
C                          :
C   24              GOTO      START
C     END           ENDSR
C     SUBRTC        BEGSR
C                          :
C                          :
C                          :
C                   ENDSR
 *
```

Figure 183. Examples of Coding Subroutines

Test Operations

The test operations are:

- "TEST (Test Date/Time/Timestamp)" on page 829
- "TESTB (Test Bit)" on page 831
- "TESTN (Test Numeric)" on page 834
- "TESTZ (Test Zone)" on page 836.

TEST is available in both the traditional syntax and free-form syntax. The other operations are available only in the traditional syntax. See Figure 195 on page 502 for an example of how %BITAND can be used to duplicate the function of TESTB.

The TESTx operations allow you to test fields specified in the result field. TEST tests for valid date, time, or timestamp data. TESTB tests the bit pattern of a result field. TESTN tests if the character field specified in the result field contain all numbers, or numbers with leading blanks, or all blanks. TESTZ tests the zone portion of the leftmost character of a character field specified in the result field. The result of these operations is indicated by the resulting indicators.

XML Operations

The XML operations include SAX parsing and reading an XML document directly into a variable.

The XML operations are:

- "XML-SAX (Parse an XML Document)" on page 886
- "XML-INTO (Parse an XML Document into a Variable)" on page 852
- "%XML (xmlDocument {:options})" on page 604
- "%HANDLER (handlingProcedure : communicationArea)" on page 539

The %HANDLER and %XML built-in functions are special built-in functions that do not return a value. They can be used only with the XML operation codes XML-SAX and XML-INTO.

XML-SAX initiates a SAX parse that repeatedly calls your SAX-handling procedure to handle events.

XML-INTO copies the information in an XML document into a program variable.

For XML documents with many repeated XML elements, it can be used to handle a limited number of XML elements at a time, having the elements passed to your XML-INTO handling procedure.

For more information about processing XML documents in your RPG programs, see *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Chapter 20. Expressions

Expressions are a way to express program logic using free-form syntax. They can be used to write program statements in a more readable or concise manner than fixed-form statements.

An expression is simply a group of operands and operations. For example, the following are valid expressions:

```
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 623
- “CHAIN (Random Retrieval from a File)” on page 633 (free-form calculations only)
- “CLEAR (Clear)” on page 642(free-form calculations only)
- “DELETE (Delete Record)” on page 655 (free-form calculations only)
- “DSPLY (Display Message)” on page 666(free-form calculations only)
- “DOU (Do Until)” on page 660
- “DOW (Do While)” on page 663
- “EVAL (Evaluate expression)” on page 676
- “EVALR (Evaluate expression, right adjust)” on page 678
- “EVAL-CORR (Assign corresponding subfields)” on page 678
- “FOR (For)” on page 692
- “IF (If)” on page 698
- “RETURN (Return to Caller)” on page 795
- “READE (Read Equal Key)” on page 777 (free-form calculations only)
- “READPE (Read Prior Equal)” on page 782 (free-form calculations only)
- “SETGT (Set Greater Than)” on page 804 (free-form calculations only)
- “SETLL (Set Lower Limit)” on page 808 (free-form calculations only)
- “SORTA (Sort an Array)” on page 815
- “WHEN (When True Then Select)” on page 843
- “XML-INTO (Parse an XML Document into a Variable)” on page 852
- “XML-SAX (Parse an XML Document)” on page 886

Figure 184 on page 478 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 184. Expression Examples

General Expression Rules

The following are general rules that apply to all expressions:

1. Expressions are coded in the Extended-Factor 2 entry on the Calculation Specification or after the operation code on a free-form calculation.
2. 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.
3. 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

4. The TRUNCNBR option (as a command parameter or as a 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

There are several types of operations:

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. For example, if **NUMBER** has the value **123.4**, the value of **-NUMBER** is **-123.4**.

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:
1. Adding two numeric values
2. Adding a duration to a date, time, or timestamp.
3. Concatenating two character, two graphic, or two UCS-2 values
4. Adding a numeric offset to a basing pointer
5. 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:
1. Subtracting two numeric values
2. Subtracting a duration from a date, time, or timestamp.
3. Subtracting a numeric offset from a basing pointer
4. 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. For example, the value of **2**3** is **8**.

= 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.

<= 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 returns '1' if both operands have the value of indicator '1'.

OR The logical or operation returns returns '1' if either operand has the value of indicator '1'.

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. Compound-assignment operators of the form op= (for example +=) combine assignment with another operation, using the target as one of the operands of the operation. The = assignment operator is used with the EVAL and EVALR operations. The op= compound-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

Built-In Functions

Built-in functions are discussed in “Built-in Functions” on page 430.

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 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 185. Using a Prototyped Procedure in an Expression

For more information on user-defined functions see “Subprocedures and Subroutines” on page 25.

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 (for example **A+B+C**) 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 492 for additional considerations.)

The following list indicates the precedence of operations from highest to lowest:
1. ()
2. Built-in functions, user-defined functions
3. unary +, unary -, NOT
4. **
5. *, /
6. binary +, binary -
7. =, <>, >, >=, <, <=
8. AND
9. OR

Figure 186 on page 482 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 186. 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

Table 72 describes the type of operand allowed for each unary operator and the type of the result. Table 73 describes the type of operands allowed for each binary operator and the type of the result. Table 74 on page 483 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 72. Types Supported for Unary Operations

Operation	Operand Type	Result Type
- (negation)	Numeric	Numeric
+	Numeric	Numeric
NOT	Indicator	Indicator

Table 73. 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

Table 73. Operands Supported for Binary Operations (continued)

Operator	Operand 1 Type	Operand 2 Type	Result Type
- (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 to 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.			
= (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 74. 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 {: Format of 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)

Table 74. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
%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
%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
%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 (unsigned)
%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
%PARMS		Numeric (integer)
%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)

Table 74. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
%SCAN	UCS-2 : UCS-2 {: Numeric}	Numeric (unsigned)
%SCANRPL	Character : Character : Character {: Numeric {: Numeric}}	Character
%SCANRPL	Graphic : Graphic : Graphic {: Numeric {: Numeric}}	Graphic
%SCANRPL	UCS-2 : UCS-2 : UCS-2 {: Numeric {: Numeric}}	UCS-2
%SECONDS	Numeric	Numeric (duration)
%SHUTDOWN		Indicator
%SQRT	Numeric	Numeric
%STATUS	{File name}	Numeric (zoned decimal)
%STR	Basing Pointer {: Numeric}	Character
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 (unsigned)
%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, Graphic, or UCS-2 {: ccsid}	Varying length UCS-2 value
%UNS	Character	Numeric (unsigned)
%UNS	Numeric	Numeric (unsigned)
%UNSH	Character	Numeric (unsigned)
%UNSH	Numeric	Numeric (unsigned)
%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)
Note: For the following built-in functions, arguments must be literals, named constants or variables.		
%PADDR	Character	Procedure or prototype pointer

Table 74. Types Supported for Built-in Functions (continued)

Operation	Operands	Result Type
%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)
%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 operations.		
%FIELDS	Any{: Any {: Any ...}	Not Applicable
%HANDLER	Prototype name : Any	Not Applicable
%KDS	Data structure {: numeric }	Not Applicable
%XML	Character or UCS-2 { : Character }	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.

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 40 digit numbers, ideally you would need a 80 digit result to hold all possible results of the multiplication. However, since RPG supports numeric values only up to 63 digits, the result is adjusted to 63 digits. In this case, as many as 17 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 Rules

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 63 digits, the precision is adjusted to fit in 63 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 behaviour 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 75 describes the default precision rules in more detail.

Table 75. Precision of Intermediate Results

Operation	Result Precision
Note: The following operations produce a numeric result. L1 and L2 are the number of digits of the two operands. Lr is the number of digits of the result. D1 and D2 are the number of decimal places of the two operands. Dr is the number of decimal places of the result. T is a temporary value.	
N1+N2	T=min (max (L1-D1, L2-D2)+1, 63) Dr=min (max (D1,D2), 63-t) Lr=t+Dr
N1-N2	T=min (max (L1-D1, L2-D2)+1, 63) Dr=min (max (D1,D2), 63-t) Lr=t+Dr
N1*N2	Lr=min (L1+L2, 63) Dr=min (D1+D2, 63-min ((L1-D1)+(L2-D2), 63))
N1/N2	Lr=63 Dr=max (63-((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+L2,16773104)
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+L2,8386552)
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 187. 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 = 63
 Dr = max(63-((L1-D1)+D2),0)
    = max(63-((5-2)+0),0)
    = max(63-3,0)
    = 60
```

2 Evaluate (Result of 1 * FLD4)

Rules:

```
Lr = min(L1+L2,63)
   = min(63+9,63)
   = 63
Dr = min(D1+D2,63-min((L1-D1)+(L2-D2),63))
   = min(60+4,63-min((63-60)+(9-4),63))
   = min(64,63-min(4+5,63))
   = min(64,55)
   = 55
```

3 Evaluate (Result of 2 + FLD5)

Rules:

```
T  = min(max(L1-D1,L2-D2)+1,63)
   = min(max(63-55,9-4)+1,63)
   = min(max(8,5)+1,63)
   = min(9,63)
   = 9
Dr = min(max(D1,D2),31-T)
   = min(max(55,4),63-9)
   = min(55,54)
   = 54
Lr = T + Dr
   = 9 + 54 = 63
```

4 Evaluate FLD2/Result of 3

Rules:

```
Lr = 63
Dr = max(63-((L1-D1)+D2),0)
   = max(63-((15-2)+ 54),0)
```

```
      = max(63-(13+54),0)
      = max(-4,0)
   ****  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 behaviour 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 behaviour 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 63).
   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 63).
   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             26p 2
D p2              s             26p 2
D p3              s             26p 2
D p4              s             26p 9
D s1              s             26s 2
D s2              s             26s 2
D i1              s             10i 0
D f1              s              8f
D proc            pr            15p 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.

 /FREE
  // Example 1:
  eval    p1 = p1 * p2 * p3;
  // p1*p2        -> P(52,4); P(52,4)
  // p1*p2*p3     -> P(78,6); P(63,0) (decimal positions are truncated)
  eval(r) p1 = p1 * p2 * p3;
  // p1*p2        -> P(52,4); P(52,4)
  // p1*p2*p3     -> P(78,6); P(63,2) (decimal positions do not drop
  //                  below target decimal positions)
  eval(rh)p1 = p1 * p2 * p3;
  // p1*p2        -> P(52,4); P(52,5)
  // p1*p2*p3     -> P(78,6); P(63,3) (decimal positions do not drop
  //                  below target decimals + 1)
  // Example 2:
  eval    p4 = p1 * p2 * proc (s1*s2*p4);
  // p1*p2        -> P(52,4); P(52,4)
  // s1*s2        -> P(52,4); P(52,4)
  // s1*s2*p4     -> P(78,13); P(63,0) (decimal positions are truncated)
  // p1*p2*proc() -> P(67,7); P(63,3)  (decimal positions are truncated)
  eval(r) p4 = p1 * p2 * proc (s1*s2*p4);
  // p1*p2        -> P(52,4); P(52,4)
  // s1*s2        -> P(52,4); P(52,4)
  // s1*s2*p4     -> P(78,13); P(63,5)
  // p1*p2*proc() -> P(67,7); P(63,7)  (we keep all decimals since we are
  //                   already below target decimals)
/END-FREE
```

Figure 188. 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 behaviour. 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 189, 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 189. Sample coding of a call with side effects

Chapter 21. Built-in Functions

This chapter describes, in alphabetical order, each built-in function.

%ABS (Absolute Value of Expression)

```
%ABS(numeric expression)
```

%ABS returns the absolute value of the numeric expression specified as the 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 434 or "Built-in Functions" on page 430.

```
 *..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 190. %ABS Example

%ADDR (Get Address of Variable)

```
%ADDR(variable)
%ADDR(varying-length variable : *DATA)
```

%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.

%ADDR returns the address of the data portion of a variable-length field when *DATA is specified as the second parameter of %ADDR.

If %ADDR with an array index parameter is specified as parameter for 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 will contain 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 191 on page 495.

If the variable specified as parameter is a table, multiple occurrence data structure, or subfield of a multiple occurrence data structure, the address will be 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 191. %ADDR Example

```
1. Use %ADDR(fld:*DATA) to call a procedure with the
   address of the data portion of a varying field.

// Assume procedure "uppercaseData" requires a pointer and a length.
// %ADDR(fld:*DATA) returns the pointer to the data portion of
// the varying field, and %LEN(fld) returns the length.
uppercaseData (%ADDR(fld : *DATA) : %LEN(fld));

2. Use %ADDR(fld:*DATA) to determine the maximum size
   of the data portion of a varying field.

// The number of bytes used for the prefix is the
// offset between the address of the field and the
// address of the data.
prefix_size = %addr(fld : *data) - %addr(fld);

// The number of bytes used for the data is the
// difference between the total bytes and the
// bytes used for the prefix.
data_size = %size(fld) - prefix_size;

// If variable "fld" is UCS-2 or DBCS, the number
// of characters is half the number of bytes
max_dbcs_chars = data_size / 2;
```

*Figure 192. Example of %ADDR with *DATA*

%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 the maximum size allowed.

The maximum size allowed depends on the type of heap storage used for RPG memory management operations due to the ALLOC keyword on the Control specification. If the module uses teraspace heap storage, the maximum size allowed is 4294967295 bytes. Otherwise, the maximum size allowed is 16776704 bytes.

The maximum size available at runtime may be less than the maximum size allowed by RPG.

For more information, see "Memory Management Operations" on page 458.

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 193. %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 194 on page 502, Figure 195 on page 502, and Figure 196 on page 503 for examples demonstrating the use of %BITAND.

For more information, see “Bit Operations” on page 439 or “Built-in Functions” on page 430.

%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 194 on page 502 for an example demonstrating the use of %BITNOT.

For more information, see “Bit Operations” on page 439 or “Built-in Functions” on page 430.

%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 194 on page 502 for an example demonstrating the use of %BITOR.

For more information, see “Bit Operations” on page 439 or “Built-in Functions” on page 430.

%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 439 or “Built-in Functions” on page 430.

Examples of Bit Operations

```
D const           c                   x'0007'

D ch1             s              4a   inz(%BITNOT(const))
 * ch1 is initialized to x'FFF84040'
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 194. 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 195. 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 196. 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 197. 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 includes the shift-in and shift-out characters. For example, if a 5 character graphic field is converted, the returned value is 12 characters (10 bytes of graphic data plus the two shift characters). 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 447 or “Built-in Functions” on page 430.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
D Name            S             20G   VARYING INZ(G'oXXYYZZi')
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 the job formats, use this:
   //-------------------------------------------------------------
   result = 'It is ' + %CHAR(time : *jobrun)
            + ' on ' + %CHAR(date : *jobrun);
   // If the job date format is *MDY- and the time separator is '.',
   // then the result = 'It is 12.23.34 on 97-02-03'

   //--------------------------------------------------------------
   // 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 oXXYYZZi.'

   //----------------------------------------------------
   // Use %CHAR to convert a number to character format:
   //----------------------------------------------------
   result = 'You have ' + %char(points) + ' points.';
   // result = 'You have 234 points.'
   //
 /END-FREE
```

Note: The graphic literal in this example is not a valid graphic literal. See "Graphic Format" on page 183 for more information.

Figure 198. %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 468 or “Built-in Functions” on page 430.

```
 *..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 199. %CHECK Example

See also Figure 201 on page 510.

%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 468 or “Built-in Functions” on page 430.

```
 *..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 200. %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 201. %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 Type” on page 206. 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 263.

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 457 or “Built-in Functions” on page 430.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

  string = '040596';
  date =  %date(string:*MDY0);
  // date now contains d'1996-04-05'
/END-FREE
```

Figure 202. %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 232 on page 555.

For more information, see “Date Operations” on page 449 or “Built-in Functions” on page 430.

%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 227 on page 548 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 263 and “TIMFMT(fmt{separator})” on page 277.

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 204 on page 515 shows an example of the %DEC built-in function.

For more information, see “Conversion Operations” on page 447 or “Built-in Functions” on page 430.

```
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 203. 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 447 or “Built-in Functions” on page 430.

%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 204. 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('$1,234,567.89')
D num             s             21p 9
 /free
    // Use the %XLATE built-in function to replace any currency
    // symbol, asterisks or thousands separators with blanks
    num = %dech(%xlate('$*,' : '   ' : 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 205. 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 206. %DECPOS Example

See Figure 227 on page 548 for an example of %DECPOS with %LEN.

For more information, see “Size Operations” on page 467 or “Built-in Functions” on page 430.

%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 difference is calculated by subtracting the second operand from the first.

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 449 or “Built-in Functions” on page 430.

```
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 207. 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 208. 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 567.)

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 434 or “Built-in Functions” on page 430.

This function is illustrated in Figure 242 on page 567.

%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 zeros 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, or the default, '$'. When *CURSYM is specified, the currency symbol is placed in the 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). In the second parameter, 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 447 or “Built-in Functions” on page 430.

```
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 209. %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:

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
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
CL0N01Factor1+++++++Opcode&ExtExtended-factor2+++++++++++++++++++++++++++
C                   EVAL      editedVal = editparens(neg)
 * Now editedVal has the value '(12.30)   '
C                   EVAL      editedVal = editparens(pos)
 * 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
 * Use parentheses if the value is negative
C                   IF        val < 0
C                   EVAL      lparen = '('
C                   EVAL      rparen = ')'
C                   ENDIF
 * Return the edited value
 * Note that the '1' edit code does not include a sign so we
 * don't have to calculate the absolute value.
C                   RETURN    lparen             +
C                             %editc(val : '1')  +
C                             rparen
P editparens      E
```

Figure 210. %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 447 or “Built-in Functions” on page 430.

```
 *..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 211. %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 447 or “Built-in Functions” on page 430.

```
 *..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 212. %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. The value returned is in unsigned integer format (type U). 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 438 or “Built-in Functions” on page 430.

```
 *..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 213. %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 772
- “READC (Read Next Changed Record)” on page 775
- “READE (Read Equal Key)” on page 777
- “READP (Read Prior Record)” on page 780
- “READPE (Read Prior Equal)” on page 782
- “WRITE (Create New Records)” on page 847 (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 633
- “OPEN (Open File for Processing)” on page 759
- “SETGT (Set Greater Than)” on page 804
- “SETLL (Set Lower Limit)” on page 808

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 453 or “Built-in Functions” on page 430.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++
 * File INFILE has record format INREC
FINFILE    IF   E             DISK

 /FREE
    READ INREC;  // read a record
    IF  %EOF;
                 // handle end of file
    ENDIF;
 /END-FREE
```

Figure 214. %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
FFILE2     IF   E             DISK

 * 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 215. %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 808
- "LOOKUP (Look Up a Table or Array Element)" on page 711

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 examples, see Figure 332 on page 713 and Figure 378 on page 811.

For more information, see "File Operations" on page 453, "Result Operations" on page 467, or "Built-in Functions" on page 430.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
 * File CUSTS has record format CUSTREC
FCUSTSIF   E           K DISK

 /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 216. %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'
     CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
     C     SearchName    LOOKUP    TabNames                           10  10
     C                   SELECT
     C                   WHEN      %EQUAL
      * An exact match was found
     C     'Exact       'DSPLY                   TabNames
     C                   WHEN      %FOUND
      * A name was found greater than SearchName
     C     'Next greater'DSPLY                   TabNames
     C                   OTHER
      * Not found.  SearchName is greater than all the names in the table
     C     'Not found   'DSPLY                   SearchName
     C                   ENDSL
     C                   RETURN
**CTDATA TabNames
Alexander
Andrea
Bohdan
Martha
Samuel
```

Figure 217. %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. The CALLP operation can also set %ERROR.

For examples of the %ERROR built-in function, see Figure 249 on page 580 and Figure 250 on page 581.

For more information, see “Result Operations” on page 467 or “Built-in Functions” on page 430.

%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.
2. The name can be a subfield from a data structure defined with the EXTNAME/LIKEREC keyword using the file/format name of the record being updated. *INPUT must be specified with the keyword used. The name specified must contain the subfield name that corresponds to the input field. For a qualified data structure, the simple qualified name of the subfield is used.
3. The name can be a subfield of a data structure defined with the LIKEDS keyword of a data structure defined as described above.

%FIELDS specifies a list of fields to update. For example:

Figure 218. 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 447 or “Built-in Functions” on page 430.

```
 *..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 219. %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'.

The operations that set %FOUND are:
- File operations:
  - “CHAIN (Random Retrieval from a File)” on page 633
  - “DELETE (Delete Record)” on page 655
  - “SETGT (Set Greater Than)” on page 804
  - “SETLL (Set Lower Limit)” on page 808
- String operations:
  - “CHECK (Check Characters)” on page 636
  - “CHECKR (Check Reverse)” on page 639
  - “SCAN (Scan String)” on page 799

  Note: Built-in function %SCAN does not change the value of %FOUND.
- Search operations:
  - “LOOKUP (Look Up a Table or Array Element)” on page 711

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 an example of %FOUND with LOOKUP, see Figure 217.

For more information, see “File Operations” on page 453, “Result Operations” on page 467, or “Built-in Functions” on page 430.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Filename+IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
 * File CUSTS has record format CUSTREC
FCUSTS     IF   E           K DISK

 /FREE
    // Check if the customer is in the file
    chain Cust CustRec;
    if %found;
       exsr HandleCustomer;
    endif;
 /END-FREE
```

Figure 220. %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
FGOLD      IF   E           K DISK

 /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 221. %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 222. %FOUND used to Test a String Operation

%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 CCSID of the job. 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 parameter is character, the character data must be in the form

```
shift-out graphic-data shift-in
```

For example, 'oAABBCCi'.

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 “Graphic Format” on page 183, “Conversion Operations” on page 447, or “Built-in Functions” on page 430.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
H*Keywords+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H ccsid (*graph: 300)

D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
D  char           S              8A   inz('oXXYYZZi')
 * The %GRAPH built-in function is used to initialize a graphic field
D  graph          S             10G   inz (%graph ('oAABBCCDDEEi'))
D  ufield         S              2C   inz (%ucs2 ('oFFGGi'))
D  graph2         S              2G   ccsid (4396) inz (*hival)
D  isEqual        S              1N
D  proc           PR
D    gparm                       2G   ccsid (4396) value

 /FREE
    graph = %graph (char) + %graph (ufield);
    // graph now has the value XXYYZZFFGG.
    // %graph(char) removes the shift characters from the
    // character data, and treats the non-shift data as
    // graphic data.

    isEqual = graph = %graph (graph2 : 300);
    // The result of the %GRAPH built-in function is the value of
    // graph2, converted from CCSID 4396 to CCSID 300.

    graph2 = graph;
    // The value of graph is converted from CCSID 300 to CCSID 4396
    // and stored in graph2.
    // This conversion is performed implicitly by the compiler.

    proc (graph);
    // The value of graph is converted from CCSID 300 to CCSID 4396
    // implicitly, as part of passing the parameter by value.
 /END-FREE
```

Figure 223. %GRAPH Examples

%HANDLER (handlingProcedure : communicationArea)

%HANDLER is used to identify a procedure to handle an event or a series of events. %HANDLER does not return a value, and it can only be specified as the first operand of XML-SAX and XML-INTO.

The first operand, *handlingProcedure* specifies the prototype of the handling procedure. The return value and parameters specified by the prototype, or by the procedure interface if the prototype is not explicitly specified, must match the parameters required for the handling procedure; the requirements are determined by the operation that %HANDLER is specified for. See "XML-SAX (Parse an XML Document)" on page 886 and "XML-INTO (Parse an XML Document into a Variable)" on page 852 for the specific requirements for the definition of the handling procedures.

The second operand, *communicationArea*, specifies a variable to be passed as a parameter on every call to the handling procedure. The operand must be an exact match for the first prototyped parameter of the handling procedure, according to the same rules that are used for checking prototyped parameters passed by reference. The communication-area parameter can be any type, including arrays and data structures.

When an operation code uses the %HANDLER built-in function, the following sequence of events occurs:

1. The operation using the %HANDLER built-in function begins.
2. When an event occurs during the operation that must be handled by the handling procedure, the RPG runtime calls the handling procedure specified as the first operand of %HANDLER. The first parameter passed to the handling procedure is the communication area that was specified as the second operand of %HANDLER. The other parameters depend on the operation and the nature of the event that occurred.
3. The handling procedure processes the parameters, possibly updating the communication-area parameter.
4. The handling procedure returns a zero if it completed successfully, and a non-zero value if it did not complete successfully.
5. If the returned value was zero, the RPG runtime continues processing until either the operation is complete, or another event occurs. If the returned value was not zero, the operation ends.
6. If another event occurs, the handling procedure is called again. If the previous call to the handling procedure changed the communication area, the changes can be seen on subsequent calls.
7. When the operation is complete, control passes to the statement following the operation that used the %HANDLER built-in function. If the handling procedure changed the communication area, the changes can be seen in the procedure that used the %HANDLER built-in function.

The communication area can be used for several purposes.

1. To communicate information from the procedure coding the %HANDLER built-in function to the handling procedure.
2. To communicate information from the handling procedure back to the procedure coding the %HANDLER built-in function.
3. To keep state information between successive calls of the handling procedure. State information can also be kept in static variables in the handling procedure,

but when static variables are used, incorrect results can occur if the handling procedure has been enabled by more than one %HANDLER operation. By using a communication area parameter, the usages of the handling procedure are independent from each other.

```
 * Data structure used as a parameter between
 * the XML-SAX operation and the handling
 * procedure.
 *   - "attrName" is set by the procedure doing the
 *     XML-SAX operation and used by the handling procedure
 *   - "attrValue" is set by the handling procedure
 *     and used by the procedure doing the XML-SAX
 *     operation
 *   - "haveAttr" is used internally by the handling
 *     procedure
D info            DS
D   attrName                    20A    VARYING
D   haveAttr                      N
D   attrValue                   20A    VARYING

 * Prototype for procedure "myHandler" defining
 * the communication-area parameter as being
 * like data structure "info"
D myHandler       PR            10I 0
D   commArea                           LIKEDS(info)
D   event                       10I 0  VALUE
D   string                        *    VALUE
D   stringLen                   20I 0  VALUE
D   exceptionId                 10I 0  VALUE
 /free
   // The purpose of the following XML-SAX operation
   // is to obtain the value of the first "companyname"
   // attribute found in the XML document.

   // The communication area "info" is initialized with
   // the name of the attribute whose value is
   // to be obtained from the XML document.
   attrName = 'companyname';

   // Start SAX processing.  The procedure "myHandler"
   // will be called for every SAX event; the first
   // parameter will be the data structure "info".
   xml-sax(e) %handler(myHandler : info) %xml(xmldoc);
   // The XML-SAX operation is complete.  The
   // communication area can be checked to get the
   // value of the attribute.
   if not %error() and attrValue <> '';
     dsply (attrName + '=' + attrValue);
   endif;
:
:
 * The SAX handling procedure "myHandler"
P myHandler       B
D                 PI            10I 0
D   comm                               LIKEDS(info)
D   event                       10I 0  VALUE
D   string                        *    VALUE
D   stringLen                   20I 0  VALUE
D   exceptionId                 10I 0  VALUE
D value           S          65535A    VARYING
D                                      BASED(string)
D ucs2value       S          16383C    VARYING
D                                      BASED(string)
D rc              S             10I 0  INZ(0)
 /free

     select;
```

Figure 224. Using a communication-area with %HANDLER (Part 1 of 2)

```
      // When the event is a "start document" event,
      // the handler can initialize any internal
      // subfields in the communication area.
      when event = *XML_START_DOCUMENT;
         comm.haveAttr = *OFF;

      // When the event is an "attribute name" event,
      // and the value of the event is the required
      // name, the internal subfield "haveAttr" is
      // set to *ON.  If the next event is an
      // attribute-value event, the value will be
      // saved in the "attrValue" subfield.
      when event = *XML_ATTR_NAME
      and  %subst(value : 1 : stringLen) = comm.attrName;
         comm.haveAttr = *ON;
         comm.attrValue = '';

      // When "haveAttr" is on, the data from any
      // attribute-value should be saved in the "attrValue"
      // string until the *XML_END_ATTR event occurs
      when comm.haveAttr;
         select;
         when event = *XML_ATTR_CHARS
         or   event = *XML_ATTR_PREDEF_REF;
            comm.attrValue +=
                %subst(value : 1 : stringLen);
         when event = *XML_ATTR_UCS2_REF;
            stringLen = stringLen / 2;
            comm.attrValue +=
                %char(%subst(ucs2value : 1 : stringLen));
         when event = *XML_END_ATTR;
            // We have the entire attribute value
            // so no further parsing is necessary.
            // A non-zero return value tells the
            // RPG runtime that the handler does
            // not want to continue the operation
            rc = -1;
        endsl;

      endsl;

    return rc;
  /end-free
 P                 E
```

Figure 224. Using a communication-area with %HANDLER (Part 2 of 2)

For more examples of %HANDLER, see “XML-SAX (Parse an XML Document)” on page 886 and “XML-INTO (Parse an XML Document into a Variable)” on page 852.

For more information, see “XML Operations” on page 475 or “Built-in Functions” on page 430.

%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 232 on page 555.

For more information, see “Date Operations” on page 449 or “Built-in Functions” on page 430.

%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 447 or “Built-in Functions” on page 430.

Figure 225 on page 545 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 447 or “Built-in Functions” on page 430.

```
 *..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 225. %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 226. Example of Search on Keyed Input/Output Operations

%LEN (Get or Set Length)

```
%LEN(expression)
%LEN(varying-length expression : *MAX)
```

%LEN can be used to get the length of a variable expression, to set the current length of a variable-length field, or to get the maximum length of a varying-length expression.

The parameter must not be a figurative constant.

For more information, see “Size Operations” on page 467 or “Built-in Functions” on page 430.

%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 227. %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, and when *MAX is not specified.

```
 *..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 228. %LEN with Variable-Length Field Example

%LEN Used to Get the Maximum Length of Varying-Length Expressions

When the second parameter of %LEN is *MAX, this function returns the maximum number of characters for a varying-length expression. When the first parameter of %LEN is a field name, this value is the same as the defined length of the field. For example, if a variable-length UCS-2 field is defined as 25C, %LEN(fld:*MAX) returns 25.

%LEN Used to Get the Maximum Length of Varying-Length Expressions

```
D char_varying    s           100a   varying
D ucs2_varying    s          5000c   varying
D graph_varying   s          7000g   varying(4)
D graph_fld10     s            10g
D char_fld10      s            10a
 /free
     // Calculate several length and size values
     // - The maximum length, %LEN(*MAX), measured in characters
     // - The current length, %LEN, measured in characters
     // - The size, %SIZE, measured in bytes, including the
     //   2- or 4-byte length prefix

     // Each alphanumeric character has one byte
     char_varying = 'abc'; // Length is 3
     max_len = %len(char_varying : *MAX);
     len = %len(char_varying);
     size = %size(char_varying);
     // max_len = 100
     // len     = 3
     // size    = 102    (100 + 2)

     // Each UCS-2 character has two bytes
     ucs2_varying = 'abc'; // Length is 3
     max_len = %len(ucs2_varying : *MAX);
     len = %len(ucs2_varying);
     size = %size(ucs2_varying);
     // max_len = 5000
     // len     = 3
     // size    = 10002    (5000 * 2 + 4)

     // Each graphic character has two bytes.
     // For field graph_varying, VARYING(4) was specified,
     // so the length prefix has four bytes
     graph_varying = graph_fld10; // Length is 10
     max_len = %len(graph_varying : *MAX);
     len = %len(graph_varying);
     size = %size(graph_varying);
     // max_len = 7000
     // len     = 10
     // size    = 14004    (7000 * 2 + 4)

     // Calculate %LEN(*MAX) of a concatenation
     graph_varying = %subst(graph_fld10:1:5); // Length is 5
     max_len = %len(graph_varying + graph_fld10 : *MAX);
     len = %len(graph_varying + graph_fld10);
     // max_len = 7010 (7000 + 10)
     // len     = 15   (5 + 10)

     // Calculate %LEN(*MAX) of a %TRIM expression
     char_fld10 = '1234'; // Trimmed length is 4
     max_len = %len(%trim(char_fld10) : *MAX);
     len = %len(%trim(char_fld10));
     // max_len = 10 (maximum trimmed length)
     // len = 4      (actual trimmed length)
```

*Figure 229. %LEN with *MAX Example*

%LOOKUPxx (Look Up an Array Element)

```
%LOOKUP(arg : array | keyed array data structure {: start_index {: number_of_elements}})
%LOOKUPLT(arg : array | keyed array data structure {: start_index {: number_of_elements}})
%LOOKUPGE(arg : array | keyed array data structure {: start_index {: number_of_elements}})
%LOOKUPGT(arg : array | keyed array data structure {: start_index {: number_of_elements}})
%LOOKUPLE(arg : array | keyed array data structure {: start_index {: number_of_elements}})
```

The following functions return the array index of the item in the array or the keyed array data structure that matches 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 value returned is in unsigned integer format (type U).

The search starts at index *start_index* and continues for *number_of_elems* elements. By default, the entire array is searched.

The second parameter can be a scalar array in the form ARRAY_NAME, or a keyed array data structure in the form ARRAY_DS_NAME(*).SUBFIELD_NAME.

To search an array data structure, specify the data structure name with an index of (*), then specify the subfield to be used as the key for the search. For example, to search for a value of 'XP2' in the CODE subfield of array data structure INFO, specify 'XP2' as the first parameter and specify INFO(*).CODE as the second parameter. The part of the qualified name up to the (*) index must represent an array, and the part of the qualified name after the (*) must represent a scalar subfield, or indexed array of scalars.

The first two parameters can have any type but must have the same type. For a keyed data structure array, the first parameter must have the same type as the key. 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. The ALTSEQ table is used, unless *arg* or *array* is defined with ALTSEQ(*NONE).

Built-in functions %FOUND and %EQUAL are not set following a %LOOKUP operation.

The %LOOKUPxx built-in 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 438
- "Built-in Functions" on page 430
- "Array Data Structures" on page 137

```
*..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 = %LOOKUPGT('Sudbury':arr);
  // n = 6

  n = %LOOKUPGT('Yorks':arr:2:4);
  // n = 0 (not found between elements 2 and 5)
/END-FREE
```

Figure 230. %LOOKUPxx with a scalar array

```
D emps           DS                   QUALIFIED DIM(20)
D    name                      25A   VARYING
D    id                         9S 0
D numEmps        S             10I 0
 /FREE
   emps(1).name = 'Mary';
   emps(1).id = 00138;
   emps(2).name = 'Patrick';
   emps(2).id = 10379;
   emps(3).name = 'Juan';
   emps(3).id = 06254;
   numEmps = 3;

   // Search for employee 'Patrick'
   n = %lookup('Patrick' : emps(*).name : 1 : numEmps);
   // n = 2

   // Search for the employee with id 06254
   n = %lookup(06254 : emps(*).id : 1 : numEmps);
   // n = 3

   // Search for employee 'Bill' (not found)
   n = %lookup('Bill' : emps(*).name : 1 : numEmps);
   // n = 0
```

Figure 231. %LOOKUP with an array data structure

Sequenced arrays that are not in the correct sequence

When the data is not in the correct sequence for a sequenced array, the %LOOKUPxx built-in functions and the LOOKUP operation code may find different values. The %LOOKUPxx built-in functions may not find a data value even if it is present in the array.

Since a binary search is used by the %LOOKUPxx built-in 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 232 on page 555.

For more information, see “Date Operations” on page 449 or “Built-in Functions” on page 430.

%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 449 or “Built-in Functions” on page 430.

```
*..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 232. %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 232 on page 555.

For more information, see "Date Operations" on page 449 or "Built-in Functions" on page 430.

%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 ALWNULL(*USRCTL) keyword is specified on a control specification or as a command parameter. 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 219 for more information on handling records with null-capable fields and keys.

For more information, see "Indicator-Setting Operations" on page 456 or "Built-in Functions" on page 430.

```
*..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 233. %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 754.

```
 *..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 234. %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 module 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 module initialization, the file is considered closed, and %OPEN returns '0'.

For more information, see “File Operations” on page 453 or “Built-in Functions” on page 430.

```
 *..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 235. %OPEN Example

%PADDR (Get Procedure Address)

```
%PADDR(string|prototype)
```

%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 constant or a prototype name. If the prototype for a procedure is implicitly defined from its procedure interface, the prototype name is the same as the procedure name.

The character constant can be a character or hexadecimal literal or constant name that represents a character or hexadecimal literal. When a character constant is used, this identifies the entry point by name.

The prototype must a prototype for a bound call. The EXTPGM keyword cannot be used. The entry point identified by the prototype is the procedure identified in the EXTPROC keyword for the prototype. If the EXTPROC keyword is not specified, the entry point is the the same as the prototype name (in upper case).

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
D
D PROC            S               *   PROCPTR
D                                     INZ (%PADDR ('FIRSTPROG'))
D PROC1           S               *   PROCPTR
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
 *
 *  The following statement calls procedure 'FIRSTPROG'.
 *
C                   CALLB     PROC
 *-----------------------------------------------------------------
 * The following statements call procedure 'NextProg'.
 * This a C procedure and is in mixed case.  Note that
 * the procedure name is case sensitive.
 *
C                   EVAL      PROC1 = %PADDR ('NextProg')
C                   CALLB     PROC1
```

Figure 236. %PADDR Example with an Entry Point

%PADDR Used with a Prototype

The argument of %PADDR can be a prototype name, with the following restrictions:

- It must not be a prototype for a Java method.
- It must not have the EXTPGM keyword.
- If its EXTPROC keyword has a procedure pointer for an argument, %PADDR cannot be used in definition specifications.

```
 *---------------------------------------------------------------
 * Several prototypes
 *---------------------------------------------------------------
D proc1           PR
D proto2          PR                    EXTPROC('proc2')
D proc3           PR                    EXTPROC(procptr3)
D pgm1            PR                    EXTPGM('PGM3')
D meth            PR                    EXTPROC(*JAVA : 'myClass'
D                                             : 'meth1')

D procptr3        S               *

 *---------------------------------------------------------------
 * Valid examples of %PADDR with prototype names as the argument
 *---------------------------------------------------------------

 * constant1 is the same as %PADDR('PROC1') since 'PROC1' is the
 * procedure called by the prototype proc1
D constant1       C                   %PADDR(proc1)

 * constant2 is the same as %PADDR('proc2') since 'proc2' is the
 * procedure called by the prototype proto2
D constant2       C                   %PADDR(proto2)

 * %paddr(proc3) is the same as procedure pointer procptr3 since
 * procptr3 points to the procedure called by prototype proc3
C                   eval      procptr = %paddr(proc3)

 *---------------------------------------------------------------
 * Examples of %PADDR with prototype names as the argument
 * that are not valid
 *---------------------------------------------------------------
 * %PADDR(pgm1) is not valid because it is a prototype for a program
 * %PADDR(meth) is not valid because it is a prototype for a Java method
```

Figure 237. %PADDR Example with a Prototype

```
 * constant1 is the same as %PADDR('myProc1').  Prototype
 * proc1 is implicitly defined from the procedure interface
 * of procedure proc1.  The external name 'myProc1' is
 * defined by the EXTPROC keyword of the implicitly defined
 * prototype.
D constant1       C                   %PADDR(proc1)

 * constant2 is the same as %PADDR('PROC2').  Prototype
 * proc2 has no prototype or procedure interface, so it has
 * a default prototype with the external name the same as
 * the internal procedure name.
D constant2       C                   %PADDR(proc2)


P proc1           B
 * The prototype for proc1 is implicitly defined from the
 * procedure interface.
 * - The name of the implicit prototype is proc1, the name
 *   of the procedure
 * - The external procedure name is 'myProc1' taken from the
 *   EXTPROC keyword of the procedure interface
D                 PI                  EXTPROC('myProc1')
 ...
P                 E

P proc2           B
 * No procedure interface is specified.
 * A default prototype is implicitly defined.
 * - The name of the implicit prototype is proc2, the name
 *   of the procedure
 * - The external procedure name is 'PROC2' taken from the
 *   uppercased form of the name of the procedure.
...
P                 E
```

Figure 238. %PADDR with procedures whose prototype is implicitly defined from the procedure interface

%PARMS (Return Number of Parameters)

%PARMS returns the number of parameters that were passed to the procedure in which %PARMS is used. For a cycle-main procedure, %PARMS is the same as *PARMS in the program status data structure.

When %PARMS is used in a procedure that was called by a bound call, the value returned by %PARMS is not available if the calling program or procedure does not pass a minimal operational descriptor. The ILE RPG compiler always passes one, but other languages do not. So if the caller is written in another ILE language, it will need to pass an operational descriptor on the call. If the operational descriptor is not passed, the value returned by %PARMS cannot be trusted. The value returned by %PARMS will be -1 if the system can determine that the operational descriptor was not passed, but in some cases when the system cannot detect this, the value returned by %PARMS may be an incorrect value that is zero or greater.

The value returned by %PARMS includes the additional first parameter that is used to handle the the return value when the RTNPARM keyword is specified. For more information, see “RTNPARM” on page 363.

For more information, see “Call Operations” on page 440 or “Built-in Functions” on page 430.

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 * Prototype for procedure MaxInt which calculates the maximum
 * value of its parameters (at least 2 parameters must be passed)
D MaxInt          PR            10I 0
D  p1                           10I 0 VALUE
D  p2                           10I 0 VALUE
D  p3                           10I 0 VALUE OPTIONS(*NOPASS)
D  p4                           10I 0 VALUE OPTIONS(*NOPASS)
D  p5                           10I 0 VALUE OPTIONS(*NOPASS)
D Fld1            S             10A   DIM(40)
D Fld2            S             20A
D Fld3            S            100A
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
C     *ENTRY        PLIST
C                   PARM                    MaxSize          10 0
 * Make sure the main procedure was passed a parameter
C                   IF        %PARMS < 1
C     'No parms'    DSPLY
C                   RETURN
C                   ENDIF
 * Determine the maximum size of Fld1, Fld2 and Fld3
C                   EVAL      MaxSize = MaxInt(%size(Fld1:*ALL) :
C                                             %size(Fld2) :
C                                             %size(Fld3))
C     'MaxSize is'  DSPLY                   MaxSize
C                   RETURN
```

Figure 239. %PARMS Example (Part 1 of 2)

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++++++++
 *---------------------------------------------------------------------
 * MaxInt - return the maximum value of the passed parameters
 *---------------------------------------------------------------------
P MaxInt          B
D MaxInt          PI            10I 0
D  p1                           10I 0 VALUE
D  p2                           10I 0 VALUE
D  p3                           10I 0 VALUE OPTIONS(*NOPASS)
D  p4                           10I 0 VALUE OPTIONS(*NOPASS)
D  p5                           10I 0 VALUE OPTIONS(*NOPASS)
D Max             S             10I 0 INZ(*LOVAL)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
CL0N01Factor1+++++++Opcode(E)+Extended-factor2++++++++++++++++++++++++++++
 * Branch to the point in the calculations where we will never
 * access unpassed parameters.
C                   SELECT
C                   WHEN      %PARMS = 2
C                   GOTO      PARMS2
C                   WHEN      %PARMS = 3
C                   GOTO      PARMS3
C                   WHEN      %PARMS = 4
C                   GOTO      PARMS4
C                   WHEN      %PARMS = 5
C                   GOTO      PARMS5
C                   ENDSL
 * Determine the maximum value.  Max was initialized to *LOVAL.
C     PARMS5        TAG
C                   IF        p5 > Max
C                   EVAL      Max = p5
C                   ENDIF
 *
C     PARMS4        TAG
C                   IF        p4 > Max
C                   EVAL      Max = p4
C                   ENDIF
 *
C     PARMS3        TAG
C                   IF        p3 > Max
C                   EVAL      Max = p3
C                   ENDIF
 *
C     PARMS2        TAG
C                   IF        p2 > Max
C                   EVAL      Max = p2
C                   ENDIF
C                   IF        p1 > Max
C                   EVAL      Max = p1
C                   ENDIF
C                   RETURN    Max
P MaxInt          E
```

Figure 239. %PARMS Example (Part 2 of 2)

%PARMNUM (Return Parameter Number)

%PARMNUM returns the number of the parameter in the parameter list. The operand for %PARMNUM is the name of a parameter defined as part of a procedure interface.

Notes:

1. A parameter defined using a *ENTRY PLIST cannot be specified as the operand for %PARMNUM.
2. The parameter must be specified the same way it appears in the procedure interface parameter list. If the parameter is an array, an index cannot be specified. If the parameter is a data structure, a subfield cannot be specified. If the parameter is a file, a record format cannot be specified.
3. If the RTNPARM keyword is coded for a procedure, the return value is handled as an additional first parameter. The other parameters have a number one higher than the apparent number. For example, if a procedure defined with RTNPARM has two parameters P1 and P2, %PARMNUM(P1) will return 2 and %PARMNUM(P2) will return 3.

For more information, see "Built-in Functions" on page 430.

```
D  myProc         pi            10A   RTNPARM OPDESC
D   companyName                 25A   OPTIONS(*VARSIZE)
D   errorCode                    1A   OPTIONS(*OMIT)
D   cityName                    25A   OPTIONS(*NOPASS)
 /free

    // test the length of companyName
    callp CEEDOD(%parmnum(companyName) : more parameters ...
               : parmlen : *omit);
    if parmlen < 25;
      // the full parameter was not passed
    endif;

    // test the presence of the omissible errorCode parameter
    callp CEETSTA(isPresent : %parmnum(errorCode) : *omit);
    if isPresent = 1;
       // errorCode was not omitted
    endif;

    // test the presence of the optional city parameter
    if %parms >= %parmnum(cityName);
       // cityName was passed
    endif;
```

Figure 240. Example of %PARMNUM

%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 heap storage pointed to by the returned pointer has the same value as the heap storage pointed to by *ptr*. If the new length is longer than the old length, the additional 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 the maximum size allowed.

The maximum size allowed depends on the type of heap storage used for RPG memory management operations due to the ALLOC keyword on the Control specification. If the module uses teraspace heap storage, the maximum size allowed is 4294967295 bytes. Otherwise, the maximum size allowed is 16776704 bytes.

The maximum size available at runtime may be less than the maximum size allowed by RPG.

The function returns a pointer to the allocated storage. This may be the same as *ptr* or different. If the %REALLOC function is successful, the original pointer value specified in the first operand should not be used.

When RPG memory management operations for the module are using single-level heap storage due to the ALLOC keyword on the Control specification, the %REALLOC built-in function can only handle pointers to single-level heap storage. When RPG memory management operations for the module are using teraspace heap storage, the %REALLOC built-in function operation can handle pointers to both single-level and teraspace heap storage.

For more information, see “Memory Management Operations” on page 458.

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);
  // Using two different pointers:
  pointer2 = %REALLOC(pointer1:500);
  pointer1 = *NULL;;
  // The returned value was assigned to
  // "pointer2", a different variable
  // from the input pointer "pointer1".
  // In this case, the value of "pointer1"
  // is no longer valid, so "pointer1" must
  // be set to *NULL to avoid using the
  // old value.
/END-FREE
```

Figure 241. %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 521.)

%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 434 or “Built-in Functions” on page 430.

```
 *..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 242. %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 468 or “Built-in Functions” on page 430.

```
 *..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 243. %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 468 or “Built-in Functions” on page 430.

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 244. %SCAN Example

%SCANRPL (Scan and Replace Characters)

```
%SCANRPL(scan string : replacement : source { : scan start  { : scan length  } )
```

%SCANRPL returns the string produced by replacing all occurrences of the scan string in the source string with the replacement string. The search for the scan string starts at the scan start position and continues for the scan length. The parts of the source string that are outside the range specified by the scan start position and the scan length are included in the result.

The first, second and third parameters must be of type character, graphic, or UCS-2. They can be in either fixed-length or variable-length format. These parameters must all be of the same type and CCSID.

The fourth parameter represents the starting position, measured in characters, where the search for the scan string should begin. If it is not specified, the starting position defaults to one. The value may range from one to the current length of the source string.

The fifth parameter represents the number of characters in the source string to be scanned. If the parameter is not specified, the length defaults to remainder of the source string starting from the start position. The value must be greater than or equal to zero, and less than or equal to the remaining length of the source string starting at the start position.

The starting position and length may be any numeric value or numeric expression with no decimal positions.

The returned value may be larger, equal to or smaller than the source string. The resulting length depends on the lengths of the scan string and the replacement string, and also on the number of times the replacement is performed. For example, assume the scan string is 'a' and the replacement string is 'bc'. If the source string is 'ada', the returned value has a length of five ('bcdbc'). If the source string is 'ddd', the returned value has a length of three ('ddd').

The returned value is varying length if the source string and replacement string have different lengths, or if any of the strings are varying length. Otherwise, the returned value is fixed length. The returned value has the same type as the source string.

Each position in the source string is scanned only once. For example, if the scan string is 'aa', and the source string is 'baaaaac', then the first match is in positions 2 and 3. The next scan begins at position 4, and finds a match in positions 4 and 5. The next scan begins at position 6, and does not find any further matches. If the replacement string is 'xy', then the returned value is 'bxyxyac'.

Tip: %SCANRPL can be used to completely remove occurrences of the scan string from the source string by specifying an empty replacement string.

For more information, see “String Operations” on page 468 or “Built-in Functions” on page 430.

```
//        ....+....1....+....2....+....3....+...
string1 = 'See NAME. See NAME run. Run NAME run.';

// 1. All occurrences of "NAME" are replaced by the
//    replacement value.  In the first case,
//    the resulting string is shorter than the source
//    string, since the replacment string is shorter
//    than the scan string. In the second case, the
//    resulting string is longer.
string2 = %ScanRpl('NAME' : 'Tom' : string1);
// string2 = 'See Tom. See Tom run. Run Tom run.'
string2 = %ScanRpl('NAME' : 'Jenny' : string1);
// string2 = 'See Jenny. See Jenny run. Run Jenny run.'

// 2. All occurrences of ** are removed from the string.
//    The replacement string, '', has zero length.
string3 = '*Hello**There**Everyone*';
string2 = %ScanRpl('**' : '' : string3);
// string2 = '*HelloThereEveryone*'

// 3. All occurrences of "NAME" are replaced by "Tom"
//    starting at position 6.  Since the first "N" of
//    the first "NAME" in the string is not part of the
//    source string that is scanned, the first "NAME"
//    is not considered replaceable.
string2 = %ScanRpl('NAME' : 'Tom' : string1 : 6);
// string2 = 'See NAME. See Tom run. Run Tom run.'

// 4. All occurrences of "NAME" are replaced by "Tom"
//    up to length 31.  Since the final "E" of
//    the last "NAME" in the string is not part of the
//    source string that is scanned, , the final "NAME"
//    is not considered replaceable.
string2 = %ScanRpl('NAME' : 'Tom' : string1 : 1 : 31);
// string2 = 'See Tom. See Tom run. Run NAME run.'

// 5. All occurrences of "NAME" are replaced by "Tom"
//    from position 10 for length 10.  Only the second
//    "NAME" value falls in that range.
string2 = %ScanRpl('NAME' : 'Tom' : string1 : 10 : 10);
// string2 = 'See NAME. See Tom run. Run NAME run.'
```

Figure 245. %SCANRPL 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 232 on page 555.

For more information, see “Date Operations” on page 449 or “Built-in Functions” on page 430.

%SHTDN (Shut Down)

```
%SHTDN
```

%SHTDN returns '1' if the system operator has requested shutdown; otherwise, it returns '0'. See “SHTDN (Shut Down)” on page 814 for more information.

For more information, see “Information Operations” on page 457 or “Built-in Functions” on page 430.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE
  // If the operator has requested shutdown, quit the
  // program.

  IF %SHTDN;
     QuitProgram();
  ENDIF;
/END-FREE
```

Figure 246. %SHTDN Example

%SIZE (Get Size in Bytes)

```
%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, not including leading and trailing shift 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).

The length returned for a null-capable field (%SIZE) is always its full length, regardless of the setting of its null indicator.

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. The system requires that pointers be placed in storage at addresses evenly divisible by 16. As a result, 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. If the array is non-contiguous due to being overlaid on a larger array, the value returned is the same as it would be if the array were contiguous; it does not include the storage between the non-contiguous array elements.

%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 467 or “Built-in Functions” on page 430.

```
 *..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 247. %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 434 or “Built-in Functions” on page 430.

```
 *..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 248. %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 453, “Result Operations” on page 467, or “Built-in Functions” on page 430.

```
*..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
   exfmt(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 = 01261;
         // Operation to unacquired device
         exsr DeviceError;

      when  %status = 01251;
         // Permanent I/O error
         exsr PermError;

      other;
         // Some other error occurred
         exsr FileError;
      endsl;
   endsr;
/END-FREE
```

Figure 249. %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                   EXFMT(E)  REC1
C                   IF        %ERROR
C                   SELECT
C                   WHEN      %STATUS = 01255
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 250. %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 the maximum allowed length for defining a character variable.

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 less than 1 or greater than the maximum length allowed, an error will occur.
2. If the pointer is not set, an error will occur.
3. If the storage addressed by the pointer is shorter than indicated by the length parameter, either
   a. An error will occur
   b. Data corruption will occur.

For more information, see “String Operations” on page 468 or “Built-in Functions” on page 430.

%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 251. %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 252. %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 253. %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. If the length specified as the second parameter of %STR is N, then at most N-1 bytes of the right-hand side can be used, since 1 byte must be reserved for the null-terminator 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 254. %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 built-in function.

For more information, see "Array Operations" on page 438 or "Built-in Functions" on page 430.

```
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 255. 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.numAlloc);
#         dynArrInfo.p = %realloc (dynArrInfo.p : dynArrInfo.current);
#         dynArrInfo.numAlloc = dynArrInfo.current;
#    endif;
#    if (oldElems < dynArrInfo.current);
          // Initialize new elements to blanks
         clear %subarr(dynArr : oldElems + 1 : dynArrInfo.current - oldElems);
     endif;
```

Figure 256. 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 “Built-in Functions” on page 430.

%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 449 or “Built-in Functions” on page 430.

```
*..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 257. %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-2data. 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 468 or “Built-in Functions” on page 430.

%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).

Figure 258 on page 589 shows an example of the %SUBST built-in function used for its value.

%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 the length is not specified then the string is referenced to its end. If the length refers to characters beyond the end of the string, then a run-time error is issued.

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 second and third parameters of %SUBST when it appears as the result of an assignment with an EVAL operation.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++
 *
 * In this example, CITY contains 'Toronto, Ontario'
 * %SUBST returns the value 'Ontario'.
 *
C     ' '           SCAN      CITY          C
C                   IF        %SUBST(CITY:C+1) = 'Ontario'
C                   EVAL      CITYCNT = CITYCNT+1
C                   ENDIF
 *
 * Before the EVAL, A has the value 'abcdefghijklmno'.
 * After the EVAL A has the value 'ab****ghijklmno'
 *
C                   EVAL      %SUBST(A:3:4) = '****'
```

Figure 258. %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 built-in 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                   EXPORT
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 259. %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 Type" on page 208. 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 277.

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 457 or "Built-in Functions" on page 430.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

   string = '12:34 PM';
   time =  %time(string:*USA);
   // time = t'12.34.00'
/END-FREE
```

Figure 260. %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 last three digits of the microsecond portion of the current system timestamp will be 000.

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 Type” on page 210.

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 457 or “Built-in Functions” on page 430.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/FREE

  string = '1960-09-29-12.34.56.000000';
  timest =  %timestamp(string);
   // timest now contains z'1960-09-29-12.34.56.000000'
/END-FREE
```

Figure 261. %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.

The ALTSEQ table is used, unless *arg* or *search-table* is defined with ALTSEQ(*NONE).

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 built-in 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 553.

For more information, see “Array Operations” on page 438 or “Built-in Functions” on page 430.

```
*..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 262. %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 468 or “Built-in Functions” on page 430.

```
 *..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 263. %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 264. 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 468 or “Built-in Functions” on page 430.

```
*..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 265. %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 468 or “Built-in Functions” on page 430.

```
 *..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 266. %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 267. Trimming characters other than blanks

%UCS2 (Convert to UCS-2 Value)

%UCS2 converts the value of the expression from character, graphic, or UCS-2 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 447 or “Built-in Functions” on page 430.

```
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
CL0N01Factor1+++++++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.
```

Note: The graphic literal in this example is not a valid graphic literal. See “Graphic Format” on page 183 for more information.

Figure 268. %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 447 or “Built-in Functions” on page 430.

Figure 269 on page 601 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 447 or “Built-in Functions” on page 430.

```
 *..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 269. %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 63 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 438 or "Built-in Functions" on page 430.

%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.

| If the first parameter is longer than the second parameter, the additional characters
| in the first parameter are ignored.

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 468 or “Built-in Functions” on page 430.

```
 *..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 270. %XLATE Example

%XML (xmlDocument {:options})

%XML is used as the second operand of the XML-SAX and XML-INTO operation codes to specify the XML document to be parsed, and the options to control how the document is parsed. %XML does not return a value, and it cannot be specified anywhere other than for the XML-SAX and XML-INTO operation codes.

The first operand specifies the document to be parsed. It can be a constant or variable character or UCS-2 expression containing either an XML document or the name of a file containing an XML document.

The second operand specifies options that control how the XML document is to be interpreted and parsed. It can be a constant or variable character expression. The value of the character expression is a list of zero or more options specified in the form

```
optionname1=value1 optionname2=value2
```

No spaces are allowed between the option name and the equal sign or between the equal sign and the value. However, any number of spaces can appear before, between or following the options. The options can be specified in any case. The following are all valid ways to specify the "doc=file" and "allowextra=yes" options for XML-INTO:

```
'doc=file allowextra=yes'
'       doc=file     allowextra=yes     '
'ALLOWEXTRA=YES DOC=FILE     '
'AllowExtra=Yes Doc=File     '
```

The following are **not** valid option strings:

Option string	The problem with the option string
'doc = file'	Spaces around the equal sign are not allowed
'allowextra'	Each option must have an equal sign and a value
'badopt=yes'	Only valid options are allowed
'allowextra=ok'	The 'allowextra' value can only be 'yes' or 'no'

The valid options and values depend on the context of the %XML built-in function. See “XML-SAX (Parse an XML Document)” on page 886 and “XML-INTO (Parse an XML Document into a Variable)” on page 852 for a complete list of valid options and values.

When an option is specified more than once, the last value specified is the value that is used. For example, if the options parameter has the value

```
'doc=file doc=string'
```

then the parser will use the value "string" for the "doc" option.

If the parser discovers an invalid option or invalid value, the operation will fail with status code 00352.

```
// The "options" parameter is omitted.  Default values are used for
// all options.  Since the default value for the "doc" option is
// always "string", the parser will correctly assume that the first
// parameter contains an XML document.
xmldocument = '<myfld>new value</myfld>';
XML-INTO myfld %XML(xmldocument);

// The "options" parameter is specified as a literal with two options.
XML-INTO myds %XML(xmldocument : 'allowmissing=yes allowextra=yes');

// The "options" parameter is specified as a variable expression
// with two options.
ccsidOpt = 'ccsid=' + %char(ccsid);
XML-SAX %HANDLER(mySaxHandler : myCommArea)
        %XML('myinfo.xml' : 'doc=file ' + ccsidOpt);
```

Figure 271. Examples of %XML

For more examples of %XML, see “XML-SAX (Parse an XML Document)” on page 886 and “XML-INTO (Parse an XML Document into a Variable)” on page 852.

For more information, see “XML Operations” on page 475 or “Built-in Functions” on page 430.

%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 232 on page 555.

For more information, see “Date Operations” on page 449 or “Built-in Functions” on page 430.

Chapter 22. Operation Codes

This chapter describes, in alphabetical order, each operation code.

ACQ (Acquire)

Free-Form Syntax	ACQ{(E)} *device-name workstn-file*

Code	Factor 1	Factor 2	Result Field	Indicators		
ACQ (E)	device- name	workstn-file		_	ER	_

The ACQ operation acquires the program device specified by *device-name* for the WORKSTN file specified by *workstn-file*. If the device is available, ACQ attaches it to the file. If it is not available or is already attached to the file, an error occurs.

To handle ACQ exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. If no error indicator or 'E' extender is specified, but the INFSR subroutine is specified, the INFSR receives control when an error/exception occurs. If no indicator, 'E' extender, or INFSR subroutine is specified, the default error/exception handler receives control when an error/exception occurs. For more information on error handling, see “File Exception/Errors” on page 79.

No input or output operation occurs when the ACQ operation is processed. ACQ may be used with a multiple device file or, for error recovery purposes, with a single device file. One program may acquire and have the device available to any called program which shares the file and allow the called program to release the device. See the section on "Multiple-Device Files" in the chapter about using WORKSTN files in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

For more information, see “File Operations” on page 453.

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. For the rules for specifying an ADD operation, see "Arithmetic Operations" on page 434.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The value 1 is added to RECNO.
C                   ADD       1             RECNO
 * The contents of EHWRK are added to CURHRS.
C                   ADD       EHWRK         CURHRS
 * The contents of OVRTM and REGHRS are added together and
 * placed in TOTPAY.
C     OVRTM         ADD       REGHRS        TOTPAY
```

Figure 272. ADD Operations

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.

Factor 1 is optional and may 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 is required and contains two subfactors. The first is a duration and may 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. You can add a year, month or day duration but not a minute duration to a date field. For list of duration codes and their short forms see “Date Operations” on page 449.

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 00:58:59.

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.

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/Errors” on page 96.

Note: The system places a 15-digit limit on durations. Adding a Duration with more than 15 significant digits will cause errors or truncation. These problems can be avoided by limiting the first subfactor in Factor 2 to 15 digits.

For more information on working with date-time fields, see “Date Operations” on page 449.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H TIMFMT(*USA) DATFMT(*MDY&)
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *
DDateconst       C                   CONST(D'12 31 92')
 *
 * Define a Date field and initialize
 *
DLoandate        S               D   DATFMT(*EUR) INZ(D'12 31 92')
DDuedate         S               D   DATFMT(*ISO)
Dtimestamp       S               Z
Danswer          S               T
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * Determine a DUEDATE which is xx years, yy months, zz days later
 * than LOANDATE.
C     LOANDATE      ADDDUR    XX:*YEARS     DUEDATE
C                   ADDDUR    YY:*MONTHS    DUEDATE
C                   ADDDUR    ZZ:*DAYS      DUEDATE
 * Determine the date 23 days later
 *
C                   ADDDUR    23:*D         DUEDATE
 * Add a 1234 microseconds to a timestamp
 *
C                   ADDDUR    1234:*MS      timestamp
 * Add 12 HRS and 16 minutes to midnight
 *
C     T'00:00 am'   ADDDUR    12:*Hours     answer
C                   ADDDUR    16:*Minutes   answer
 * Subtract 30 days from a loan due date
 *
C                   ADDDUR    -30:*D        LOANDUE
```

Figure 273. ADDDUR Operations

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 the maximum size supported. If the value is out of range at runtime, an error will occur with status 425. 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 maximum size allowed depends on the type of heap storage used for memory management operations due to the ALLOC keyword on the Control specification. If it is known at compile time that the module uses the teraspace storage model for memory management operations, the maximum size allowed is 4294967295 bytes. Otherwise, the maximum size allowed is 16776704 bytes.

The maximum size available at runtime may be less than the maximum size allowed by RPG.

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/Errors” on page 96.

For more information, see “Memory Management Operations” on page 458.

```
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 274. ALLOC Operation

ANDxx (And)

Free-Form Syntax	(not allowed - use the AND operator)

Code	Factor 1	Factor 2	Result Field	Indicators		
ANDxx	Comparand	Comparand				

This operation must immediately follow a ANDxx, DOUxx, DOWxx, IFxx, ORxx, or WHENxx operation. 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.

The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry must be the same as the control level entry for the associated DOUxx, DOWxx, IFxx, or WHENxx operation. Conditioning indicator entries (positions 9 through 11) are not permitted.

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. See “Compare Operations” on page 445.

For more information, see “Structured Programming Operations” on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * If ACODE is equal to A and indicator 50 is on, the MOVE
 * and WRITE operations are processed.
C     ACODE         IFEQ      'A'
C     *IN50         ANDEQ     *ON
C                   MOVE      'A'           ACREC
C                   WRITE     RCRSN
 * If the previous conditions were not met but ACODE is equal
 * to A, indicator 50 is off, and ACREC is equal to D, the
 * 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 275. ANDxx Operations

BEGSR (Beginning of 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 an RPG IV subroutine. *subroutine-name* is the subroutine name. You may specify the same name as the *subroutine-name* on the EXSR operation referring to the subroutine, in the result field of the CASxx operation referring to the subroutine, or in the entry of an INFSR file specification keyword of the subroutine is a file-error subroutine. The control level entry (positions 7 and 8) can be SR or blank. Conditioning indicator entries are not permitted.

Every subroutine must have a unique symbolic name. The keyword *PSSR used in factor 1 specifies that this is a program exception/error subroutine to handle program-detected exception/errors. Only one subroutine can be defined by this keyword. *INZSR in factor 1 specifies a subroutine to be run during the initialization step. Only one subroutine can be defined *INZSR.

See Figure 183 on page 474 for an example of coding subroutines; see “Subroutine Operations” on page 472 for general information on subroutine operations.

BITOFF (Set Bits Off)

Free-Form Syntax	(not allowed - use the%BITAND and %BITNOT built-in functions. See Figure 196 on page 503.)

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. Therefore, when using BITOFF to format a character, you should use both BITON and BITOFF: BITON to specify the bits to be set on (=1), and BITOFF to specify the bits to be set off (=0). Unless you explicitly set on or off all the bits in the character, you might not get the character you want.

If you want to assign a particular bit pattern to a character field, use the "MOVE (Move)" on page 720 operation with a hexadecimal literal in factor 2.

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:* 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 off in the result field; bits that are off do not affect the result.
- *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 set off in the result field; bits that are off are not affected.
- *Named constant:* 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 439.

```
 *    Set off bits 0,4,6 in  FieldG.  Leave bits 1,2,3,5,7 unchanged.
 *    Setting off bit 0, which is already off, results in bit 0 remaining off.
 *       Factor 2 = 10001010
 *       FieldG   = 01001111  (before)
 *       FieldG   = 01000101  (after)
C                   BITOFF    '046'         FieldG
 *    Set off bits 0,2,4,6 in FieldI.  Leave bits 1,3,5,7 unchanged.
 *    Setting off bit 2, which is already off, results in bit 2 remaining off.
 *       Factor 2 = 10101010
 *       FieldI   = 11001110  (before)
 *       FieldI   = 01000100  (after)
C                   BITOFF    BITNC         FieldI
 *    HEXNC is equivalent to literal '4567', bit pattern 00001111.
 *    Set off bits 4,5,6,7 in  FieldK.  Leave bits 0,1,2,3 unchanged.
 *       Factor 2 = 11110000
 *       FieldK   = 10000000  (before)
 *       FieldK   = 00000000  (after)
C                   BITOFF    HEXNC2        FieldK
C                   RETURN
```

Figure 276. BITOFF Example

BITON (Set Bits On)

Free-Form Syntax	(not allowed - use the %BITOR built-in function. See Figure 196 on page 503.)

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. Therefore, when using BITON to format a character, you should use both BITON and BITOFF: BITON to specify the bits to be set on (=1), and BITOFF to specify the bits to be set off (=0). Unless you explicitly set on or off all the bits in the character, you might not get the character you want.

If you want to assign a particular bit pattern to a character field, use the "MOVE (Move)" on page 720 operation with a hexadecimal literal in factor 2.

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:* 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 439.

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D FieldA          S              1A   INZ(X'00')
D FieldB          S              1A   INZ(X'00')
D FieldC          S              1A   INZ(X'FF')
D FieldD          S              1A   INZ(X'C0')
D FieldE          S              1A   INZ(X'C0')
D FieldF          S              1A   INZ(X'81')
D FieldG          S              1A   INZ(X'4F')
D FieldH          S              1A   INZ(X'08')
D FieldI          S              1A   INZ(X'CE')
D FieldJ          S              1A   INZ(X'80')
D FieldK          S              1A   INZ(X'80')
D BITNC           C                   CONST('0246')
D HEXNC           C                   CONST(X'0F')
D HEXNC2          C                   CONST(X'F0')
C*0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *    Set on bits 0,4,5,6,7 in  FieldA.  Leave bits 1,2,3 unchanged.
 *       Factor 2 = 10001111
 *       FieldA   = 00000000  (before)
 *       FieldA   = 10001111  (after)
C                   BITON     '04567'       FieldA
 *    Set on bit 3 in FieldB.  Leave bits 0,1,2,4,5,6,7 unchanged.
 *       Factor 2 = 00010000
 *       FieldB   = 00000000  (before)
 *       FieldB   = 00010000  (after)
C                   BITON     '3'           FieldB
 *    Set on bit 3 in FieldC.  Leave bits 0,1,2,4,5,6,7 unchanged.
 *    Setting on bit 3, which is already on, results in bit 3 remaining on.
 *       Factor 2 = 00010000
 *       FieldC   = 11111111  (before)
 *       FieldC   = 11111111  (after)
C                   BITON     '3'           FieldC
 *    Set on bit 3 in FieldD.  Leave bits 0,1,2,4,5,6,7 unchanged.
 *       Factor 2 = 00010000
 *       FieldD   = 11000000  (before)
 *       FieldD   = 11010000  (after)
C                   BITON     '3'           FieldD
 *    Set on bits 0 and 1 in FieldF.  Leave bits 2,3,4,5,6,7 unchanged.
 *    (Setting on bit 0, which is already on, results in bit 0 remaining on.)
 *       Factor 2 = 11000000
 *       FieldF   = 10000001  (before)
 *       FieldF   = 11000001  (after)
C                   BITON     FieldE        FieldF
 *    X'C1' is equivalent to literal '017', bit pattern 11000001.
 *    Set on bits 0,1,7 in  FieldH.  Leave bits 2,3,4,5,6 unchanged.
 *       Factor 2 = 11000001
 *       FieldH   = 00001000  (before)
 *       FieldH   = 11001001  (after)
C                   BITON     X'C1'         FieldH
 *    HEXNC is equivalent to literal '4567', bit pattern 00001111.
 *    Set on bits 4,5,6,7 in  FieldJ.  Leave bits 0,1,2,3 unchanged.
 *       Factor 2 = 00001111
 *       FieldJ   = 10000000  (before)
 *       FieldJ   = 10001111  (after)
C                   BITON     HEXNC         FieldJ
C                   RETURN
```

Figure 277. BITON Example

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 (positions 71-76) are set accordingly, and the program continues with the next operation in the sequence.

You can specify conditioning indicators. 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 label specified in the result field must be associated with a unique TAG operation and must be a unique symbolic name.

A CABxx operation in the cycle-main procedure can specify a branch:
• To a previous or a succeeding specification line
• From a detail calculation line to another detail calculation line
• From a total calculation line to another total calculation line
• From a detail calculation line to a total calculation line
• From a subroutine to a detail calculation line or a total calculation 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.

> **Attention!**
> Branching from one point in the logic to another may result in an endless loop. You must ensure that the logic of your program or procedure does not produce undesirable results.

Resulting indicators are optional. When specified, they are set to reflect the results of the compare operation. For example, the HI indicator is set when F1>F2, LO is set when F1<F2, and EQ is set when F1=F2.

See "Compare Operations" on page 445 for the rules for comparing factor 1 with factor 2.

For more information, see "Branching Operations" on page 439.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *        The field values are:
 *        FieldA = 100.00
 *        FieldB = 105.00
 *        FieldC = ABC
 *        FieldD = ABCDE
 *
 *         Branch to TAGX.
C     FieldA        CABLT     FieldB        TAGX
 *
 *         Branch to TAGX.
C     FieldA        CABLE     FieldB        TAGX
 *
 *         Branch to TAGX; indicator 16 is off.
C     FieldA        CABLE     FieldB        TAGX                     16
 *
 *         Branch to TAGX; indicator 17 is off, indicator 18 is on.
C     FieldA        CAB       FieldB        TAGX                   1718
 *
 *         Branch to TAGX; indicator 19 is on.
C     FieldA        CAB       FieldA        TAGX                      19
 *
 *         No branch occurs.
C     FieldA        CABEQ     FieldB        TAGX
 *
 *         No branch occurs; indicator 20 is on.
C     FieldA        CABEQ     FieldB        TAGX                    20
 *
 *         No branch occurs; indicator 21 is off.
C     FieldC        CABEQ     FieldD        TAGX                 21
C                   :
C     TAGX          TAG
```

Figure 278. CABxx Operations

CALL (Call a 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	LR

The CALL operation passes control to the program specified in factor 2.

Factor 2 must contain a character entry specifying the name of the program to be called.

In the result field, specify parameters in one of the following ways:
- Enter the name of a PLIST
- Leave the result field blank. This is valid if the called program does not access parameters or if the PARM statements directly follow the CALL operation.

Positions 71 and 72 must be blank.

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/Errors" on page 96.

Any valid resulting indicator can be specified in positions 75 and 76 to be set on if
the called program is an RPG program or cycle-main procedure that returns with
the LR indicator on.

Note: The LR indicator is not allowed in a thread-safe environment.

For more information on call operations, see "Call Operations" on page 440.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *  The CALL operation calls PROGA and allows PROGA to access
 *  FieldA and FieldB, defined elsewhere. PROGA is run using the content
 *  of FieldA and FieldB.  When PROGA has completed, control
 *  returns to the statement following the last PARM statement.
 *
 *
C                   CALL      'PROGA'
C                   PARM                    FieldA
C                   PARM                    FieldB
```

Figure 279. CALL Operation

CALLB (Call a Bound Procedure)

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	LR

The CALLB operation is used to call bound procedures written in any of the ILE languages.

The operation extender D may be used to include operational descriptors. This is similar to calling a prototyped procedure with CALLP when its parameters have been defined with keyword OPDESC. (Operational descriptors provide the programmer with run-time resolution of the exact attributes of character or graphic strings passed (that is, length and type of string). For more information, see chapter on calling programs and procedures in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Factor 2 is required and must be a literal or constant containing the name of the procedure to be called, or a procedure pointer containing the address of the procedure to be called. All references must be able to be resolved at bind time. The procedure name provided is case sensitive and may contain more than 10 characters, but no more than 255. If the name is longer than 255, it will be truncated to 255. The result field is optional and may contain a PLIST name.

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/Errors" on page 96.

An indicator specified in positions 75-76 will be set on when the call ends with LR set on.

Note: The LR indicator is not allowed in a thread-safe environment.

For more information on call operations, see "Call Operations" on page 440.

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 * Define a procedure pointer
D
D ProcPtr         S               *   PROCPTR INZ(%PADDR('Create_Space'))
D Extern          S             10
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * The following call linkage would be STATIC
C                   CALLB     'BOUNDPROC'
 * The following call linkage would be DYNAMIC
C                   CALL      Extern
 * The following call linkage would be STATIC, using a procedure pointer
C                   CALLB     ProcPtr
```

Figure 280. CALLB Operation

CALLP (Call a Prototyped Procedure or Program)

Free-Form Syntax	{CALLP{(EMR)}} *name*({*parm1*{:*parm2*...}})

Code	Factor 1	Extended Factor 2
CALLP (E M/R)		name{ (parm1 {:parm2...}) }

The CALLP operation is used to call prototyped procedures or programs.

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, and if the prototype does not have the same name as an operation code.

The compiler then uses the prototype name to obtain an external name, if required, for the call. If the keyword EXTPGM is specified on the prototype, the call will be a dynamic external call; otherwise it will be a bound procedure call.

If the called program or procedure is defined in a different module, a prototype for the program or procedure being called must be included in the definition specifications preceding the CALLP. If the called program or procedure is defined in the same module as the call, an explicit prototype is not required; the prototype can be implicitly defined from the procedure interface of the called program or procedure.

Note that 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 handle CALLP exceptions (program status codes 202, 211, or 231), the operation code extender 'E' can be specified. For more information on error handling, see “Program Exception/Errors” on page 96.

Note: The E extender is only active during the final call for CALLP. If an error occurs on a call that is done as part of the parameter processing, control will not pass to the next operation. For example, if FileRecs is a procedure returning a numeric value, and an error occurs when FileRecs is called in the following statement, the E extender would have no effect.

```
CALLP(E) PROGNAME(FileRecs(Fld) + 1)
```

For more information on call operations, see “Call Operations” on page 440. For more information on defining prototypes, see “Prototypes and Parameters” on page 153. For information on how operation extenders M and R are used, see “Precision Rules for Numeric Operations” on page 486.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *-------------------------------------------------------------
 *  This prototype for QCMDEXC defines two parameters:
 *   1- a character field that may be shorter in length
 *      than expected
 *   2- any numeric field
 *-------------------------------------------------------------
D qcmdexc         PR                  extpgm('QCMDEXC')
D   cmd                        200A   options(*varsize) const
D   cmdlen                      15P 5 const

 /FREE
      qcmdexc ('WRKSPLF' : %size ('WRKSPLF'));
 /END-FREE
```

Figure 281. Calling a Prototyped Program Using CALLP

```
 * 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 282. Passing an array parameter using CALLP

The following example of CALLP is from the service program example in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*. CvtToHex is a procedure in a service program created to hold conversion routines. CvtToHex converts an input string to its hexadecimal form. The prototyped calls are to the ILE CEE API, CEEDOD (Retrieve Operational Descriptor). It is used to determine the length of the input string.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *=================================================================*
 * CvtToHex - convert input string to hex output string            *
 *=================================================================*
D/COPY MYLIB/QRPGLESRC,CVTHEXPR

 *-----------------------------------------------------------------*
 * Main entry parameters                                           *
 * 1. Input:   string                   character(n)               *
 * 2. Output:  hex string               character(2 * n)           *
 *-----------------------------------------------------------------*
D CvtToHex        PI                    OPDESC
D   InString                 16383      CONST OPTIONS(*VARSIZE)
D   HexString                32766      OPTIONS(*VARSIZE)

 *-----------------------------------------------------------------*
 * Prototype for CEEDOD (Retrieve operational descriptor)          *
 *-----------------------------------------------------------------*
D CEEDOD          PR
D                               10I 0 CONST
D                               10I 0
D                               10I 0
D                               10I 0
D                               10I 0
D                               10I 0
D                               12A   OPTIONS(*OMIT)

 * Parameters passed to CEEDOD
D ParmNum         S             10I 0
D DescType        S             10I 0
D DataType        S             10I 0
D DescInfo1       S             10I 0
D DescInfo2       S             10I 0
D InLen           S             10I 0
D HexLen          S             10I 0
```

Figure 283. Calling a Prototyped Procedure Using CALLP (Part 1 of 3)

```
 *-----------------------------------------------------------------*
 * Other fields used by the program                                *
 *-----------------------------------------------------------------*
D HexDigits       C                   CONST('0123456789ABCDEF')
D IntDs           DS
D   IntNum                       5I 0 INZ(0)
D   IntChar                      1    OVERLAY(IntNum:2)
D HexDs           DS
D   HexC1                        1
D   HexC2                        1
D InChar          S              1
D Pos             S              5P 0
D HexPos          S              5P 0

 /FREE
   //-------------------------------------------------------------//
   // Use the operational descriptors to determine the lengths of //
   // the parameters that were passed.                            //
   //-------------------------------------------------------------//
   CEEDOD (1 : DescType : DataType :
               DescInfo1 : DescInfo2 : Inlen : *OMIT);
   CEEDOD (2 : DescType : DataType :
               DescInfo1 : DescInfo2 : HexLen : *OMIT);

   //-------------------------------------------------------------//
   // Determine the length to handle (minimum of the input length //
   // and half of the hex length)                                 //
   //-------------------------------------------------------------//
   if InLen > HexLen / 2;
      InLen = HexLen / 2;
   endif;

   //-------------------------------------------------------------//
   // For each character in the input string, convert to a 2-byte //
   // hexadecimal representation (for example, '5' --> 'F5')      //
   //-------------------------------------------------------------//
   HexPos = 1;
   for Pos = 1 to InLen;
       InChar = %SUBST(InString : Pos :1);
       exsr GetHex;
       %subst (HexString: HexPos: 2) = HexDs;
       HexPos = HexPos + 2;
   endfor;

   //------------------------------//
   // Done; return to caller.      //
   //------------------------------//
   return;

   //================================================================//
   // GetHex - subroutine to convert 'InChar' to 'HexDs'             //
   //                                                               //
   // Use division by 16 to separate the two hexadecimal digits.     //
   // The quotient is the first digit, the remainder is the second.  //
   //================================================================//
   begsr GetHex;
      IntChar = InChar;

      //-----------------------------------------------------//
      // Use the hexadecimal digit (plus 1) to substring the //
      // list of hexadecimal characters '012...CDEF'.        //
      //-----------------------------------------------------//
      HexC1 = %subst (HexDigits: %div(IntNum:16) + 1: 1);
      HexC2 = %subst (HexDigits: %rem(IntNum:16) + 1: 1);
```

Figure 283. Calling a Prototyped Procedure Using CALLP (Part 2 of 3)

```
   endsr;  // GetHex
/END-FREE
```

Figure 283. Calling a Prototyped Procedure Using CALLP (Part 3 of 3)

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 allows you 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.

You can specify conditioning indicators. Factor 1 and factor 2 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 (blanks are valid only if xx is blank and no resulting indicators are specified in positions 71 through 76). If factor 1 and factor 2 are not blanks, both must be of the same data type. In a CASƀƀ operation, factor 1 and factor 2 are required only if resulting indicators are specified in positions 71 through 76.

The result field must contain the name of a valid RPG IV subroutine, including *PSSR, the program exception/error subroutine, and *INZSR, the program initialization subroutine. 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. A CAS group can contain only CASxx operations. An ENDCS operation must follow the last CASxx operation to denote the end of the CAS group. After the subroutine is processed, the program continues with the next operation to be processed following the ENDCS operation, unless the subroutine passes control to a different operation.

The CASƀƀ 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 CASƀƀ operation. Any CASxx operations that follow an unconditional CASƀƀ operation in the same CAS group are never tested. Therefore, the normal placement of the unconditional CASƀƀ operation is after all other CASxx operations in the CAS group.

You cannot use conditioning indicators on the ENDCS operation for a CAS group.

See “Compare Operations” on page 445 or “Subroutine Operations” on page 472 for further rules for the CASxx operation.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The CASGE operation compares FieldA with FieldB.  If FieldA is
 *  greater than or equal to FieldB, Subr01 is processed and the
 *  program continues with the operation after the ENDCS operation.
 *
C     FieldA        CASGE     FieldB        Subr01
 *
 *  If FieldA is not greater than or equal to FieldB, the program
 *  next compares FieldA with FieldC.  If FieldA is equal to FieldC,
 *  SUBR02 is processed and the program continues with the operation
 *  after the ENDCS operation.
 *
C     FieldA        CASEQ     FieldC        Subr02
 *
 *  If FieldA is not equal to FieldC, the CAS operation causes Subr03
 *  to be processed before the program continues with the operation
 *  after the ENDCS operation.
 *  The CAS statement is used to provide a subroutine if none of
 *  the previous CASxx operations have been met.
 *
C                   CAS                     Subr03
 *
 *  The ENDCS operation denotes the end of the CAS group.
 *
C                   ENDCS
```

Figure 284. CASxx Operation

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 no factor 1 is specified, factor 2 is concatenated to the end of the result field string.

Factor 1 can contain a string, which can be one of: a field name, array element, named constant, data structure name, table name, or literal. If factor 1 is not specified, the result field is used. In the following discussion, 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'40', while for UCS-2 data a blank is x'0020'. The string portion can contain one of: a field name, array element, named constant, data structure name, table name, literal, or data structure subfield name. The number of blanks portion must be numeric with zero decimal positions, and can contain one of: 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. Then N blanks are added following the last non-blank character. Then factor 2 is 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 one of: 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. If the result field is variable-length, its length does not change.

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.

For more information, see “String Operations” on page 468.

Note: 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.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following example shows leading blanks in factor 2.  After
 * the CAT, the RESULT contains 'MR.ƀSMITH'.
 *
C                   MOVE      'MR.'         NAME              3
C                   MOVE      ' SMITH'      FIRST             6
C     NAME          CAT       FIRST         RESULT            9
 *
 *  The following example shows the use of CAT without factor 1.
 *  FLD2 is a 9 character string.  Prior to the concatenation, it
 *  contains 'ABCƀƀƀƀƀƀ'; FLD1 contains 'XYZ
 *  After the concatenation, FLD2 contains 'ABCƀƀXYZƀ'.
 *
C                   MOVEL(P) 'ABC'          FLD2              9
C                   MOVE     'XYZ'          FLD1              3
C                   CAT      FLD1:2         FLD2
```

Figure 285. CAT Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * CAT concatenates LAST to NAME and inserts one blank as specified
 * in factor 2.  TEMP contains 'Mr.ƀSmith'.
C                   MOVE      'Mr.   '      NAME              6
C                   MOVE      'Smith '      LAST              6
C     NAME          CAT       LAST:1        TEMP              9
 *
 * CAT concatenates 'RPG' to STRING and places 'RPG/400' in TEMP.
C                   MOVE     '/400'         STRING            4
C     'RPG'         CAT      STRING         TEMP              7
 *
 * The following example is the same as the previous example except
 * that TEMP is defined as a 10 byte field.  P operation extender
 * specifies that blanks will be used in the rightmost positions
 * of the result field that the concatenation result, 'RPG/400',
 * does not fill.  As a result, TEMP contains 'RPG/400ƀƀƀ'
 * after concatenation.
C                   MOVE      *ALL'*'       TEMP             10
C                   MOVE      '/400'        STRING            4
C     'RPG'         CAT(P)    STRING        TEMP
 *
 * After this CAT operation, the field TEMP contains 'RPG/4'.
 * Because the field TEMP was not large enough, truncation occurred.
C                   MOVE      '/400'        STRING            4
C     'RPG'         CAT       STRING        TEMP              5
 *
 * Note that the trailing blanks of NAME are not included because
 * NUM=0.  The field TEMP contains 'RPGIVƀƀƀƀƀ'.
C                   MOVE      'RPG  '       NAME              5
C                   MOVE      'IV   '       LAST              5
C                   Z-ADD     0             NUM               1 0
C     NAME          CAT(P)    LAST:NUM      TEMP             10
```

Figure 286. CAT Operation with leading blanks

```
*...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'oAACCBBGGi')
D Graffld2                       10G   INZ
D Graffld3                       10G   INZ(G'oAABBCCDDEEFFGGHHi')
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
 * The value 2 represents 2 graphic blanks as separators
C     G'oaai'       cat       Graffld:2     Graffld2
C                   cat       Graffld       Graffld3
```

Figure 287. 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), sets a record identifying indicator on (if specified on the input specifications), and places the data from the record into the input fields.

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.

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 546. 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 specifies the file or record format name that is to be read. A record format name is valid with an externally described file. If a file name is specified in *name* and access is by key, the CHAIN operation retrieves the first record that matches the search argument.

If *name* is a record format name and access is by key, the CHAIN operation retrieves the first record of the specified record type whose key matches the search argument. If no record is found of the specified record type that matches the search argument, a no-record-found condition exists.

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 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

For a WORKSTN file, the CHAIN operation retrieves a subfile record.

For a multiple device file, you must specify a record format in the *name* operand. Data is read from the program device identified by the field name specified in the "DEVID(fieldname)" on page 293 keyword in the file specifications for the device file. If the keyword is not specified, data is read from the device for the last successful input operation to the file.

If the file is specified as an input DISK file, all records are read without locks and so no operation extender can be specified. If the file is specified as update, all records are locked if the N operation extender is not specified.

If you are reading from an update disk file, you can specify an N operation extender to indicate that no lock should be placed on the record when it is read (e.g. CHAIN (N)). See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

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 79.

Positions 75 and 76 must be blank.

When the CHAIN operation is successful, the file specified in *name* is positioned such that a subsequent read operation retrieves the record logically following or preceding the retrieved record. When the CHAIN operation is not completed successfully (for example, an error occurs or no record is found), the file specified in *name* must be repositioned (for example, by a CHAIN or SETLL operation) before a subsequent read operation can be done on that file.

If an update (on the calculation or output specifications) is done on the file specified in *name* immediately after a successful CHAIN operation to that file, the last record retrieved is updated.

See "Database Null Value Support" on page 219 for information on handling records with null-capable fields and keys.

For more information, see "File Operations" on page 453.

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 288. CHAIN Operation with a File Name

```
FFilename++IPEASF.....L.....A.Device+.Keywords+++++++++++++++++++++++++
FCUSTFILE   IF   E           K DISK
 /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 289. CHAIN Operation Using a List of Key Fields

```
FFilename++IPEASF.....L.....A.Device+.Keywords+++++++++++++++++++++++++
FCUSTFILE   IF   E           K DISK
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 290. CHAIN Operation Using a Data Structure with an Externally-Described File

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.

You can specify a start position in factor 2, separating it from the base string by a colon. The start position is optional and defaults to 1. 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.

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 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.

Factor 1 must be a string, and can contain one of: 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 location. The base string portion of factor 2 can contain: a field name, array element, named constant, data-structure name, literal, or table name. The start location portion 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 location is specified, a value of 1 is used.

The result field can be a numeric variable, numeric array element, numeric table name, or numeric array. Define the field or array specified with no decimal positions. If graphic or UCS-2 data is used, the result field will contain double-byte character positions (that is, position 3, the 3rd double-byte character, will be character position 5).

Note: 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.

Any valid indicator can be specified in positions 7 to 11.

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/Errors" on page 96.

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.

For more information, see "String Operations" on page 468.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 * In this example, the result will be N=6, because the start
 * position is 2 and the first nonnumeric character found is the '.'.
 * The %FOUND built-in function is set to return '1', because some
 * nonnumeric characters were found.
 *
D
D Digits         C                    '0123456789'
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C
C                   MOVE      '$2000.'      Salary
C     Digits        CHECK     Salary:2      N
C                   IF        %FOUND
C                   EXSR      NonNumeric
C                   ENDIF
 *
 * Because factor 1 is a blank, CHECK indicates the position
 * of the first nonblank character.  If STRING contains 'bbbth
 * NUM will contain the value 4.
 *
C
C           ' '     CHECK     String        Num               2 0
```

Figure 291. CHECK Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 * The following example checks that FIELD contains only the letters
 * A to J.  As a result, ARRAY=(136000) after the CHECK operation.
 * Indicator 90 turns on.
 *
D
D Letter          C                   'ABCDEFGHIJ'
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
C
C                   MOVE      '1A=BC*'      Field             6
C     Letter        CHECK     Field         Array                    90
C
 *
 * In the following example, because FIELD contains only the
 * letters A to J, ARRAY=(000000).  Indicator 90 turns off.
 *
C
C                   MOVE      'FGFGFG'      Field             6
C     Letter        CHECK     Field         Array                    90
C
C
```

Figure 292. CHECK Operation

```
*...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'oDDBBCCDDi')
D Num                             5 0
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
C
C
C     G'oAABBCCi'   check     Graffld:2     Num
```

Figure 293. 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 (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 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.

You can specify a start position in factor 2, separating it from the base string by a colon. The start position is optional and defaults to the length of the string. The value in the result field is relative to the leftmost position in the source string, regardless of the start position.

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 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.

Factor 1 must be a string and can contain one of: 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 location. The base string portion of factor 2 can contain: a field name, array element, named constant, data structure name, data structure subfield name, literal, or table name. The start location portion 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 location is specified, the length of the string is used.

The result field can be a numeric variable, numeric array element, numeric table name, or numeric array. Define the field or array specified with no decimal positions. If graphic or UCS-2 data is used, the result field will contain double-byte character positions (that is, position 3, the 3rd double-byte character, will be character position 5).

Note: 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.

Any valid indicator can be specified in positions 7 to 11.

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/Errors” on page 96.

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.

For more information, see “String Operations” on page 468.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * Because factor 1 is a blank character, CHECKR indicates the
 * position of the first nonblank character.  This use of CHECKR
 * allows you to determine the length of a string.  If STRING
 * contains 'ABCDEF   ', NUM will contain the value 6.
 * If an error occurs, %ERROR is set to return '1' and
 * %STATUS is set to return status code 00100.
 *
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
```

Figure 294. CHECKR Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *
 * After the following example, N=1 and the found indicator 90
 * is on. Because the start position is 5, the operation begins
 * with the rightmost 0 and the first nonnumeric found is the '$'.
 *
D Digits          C                   '0123456789'
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C                   MOVE      '$2000.'      Salary          6
C     Digits        CHECKR    Salary:5      N                        90
C
```

Figure 295. CHECKR Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *
 * The following example checks that FIELD contains only the letters
 * A to J.  As a result, ARRAY=(876310) after the CHECKR operation.
 * Indicator 90 turns on. %FOUND would return '1'.
D
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
D Array           S              1    DIM(6)
D Letter          C                   'ABCDEFGHIJ'
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
C                   MOVE      '1A=BC***'    Field             8
C     Letter        CHECKR    Field         Array                    90
C
```

Figure 296. CHECKR Operation

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)			

The CLEAR operation sets elements in a structure (record format, data structure, array, or table) or a variable (field, subfield, array element or indicator), to their default initialization value depending on field type (numeric, character, graphic, UCS-2, indicator, pointer, or date/time/timestamp). For the default initialization value for a data type, see Chapter 9, "Data Types and Data Formats," on page 179.

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.

See "Initialization Operations" on page 457.

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

*NOKEY is optional. If *NOKEY is specified, then key fields are not cleared to their initial values.

*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. For WORKSTN file record formats (positions 36-42 on a file-description specification), if *ALL is not specified, only those fields with a usage of output or both are affected. All field-conditioning indicators of the record format are affected by the operation. When the RESET operation is applied to a record format name, and INDARA has been specified in the DDS, the indicators in the record format are not cleared.

Fields in DISK, SEQ, 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.

A RESET operation of a record format with *ALL specified is not valid when:
- A field is defined externally as input-only, and the record was not used for input.
- A field is defined externally as output-only, and the record was not used for output.
- A field is defined externally as both input and output capable, and the record was not used for either input or output.

For more information, see "Initialization Operations" on page 457.

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.

CLEAR Examples

- Figure 297 on page 644 shows an example of the CLEAR operation.
- Figure 298 on page 645 shows an example of the field initialization for the CLEAR record format.
- The examples in "RESET Examples" on page 790 also apply to CLEAR, except for the actual operation performed on the fields.

```
 *..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 297. CLEAR Operation

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
A* Field2 and Field3 are defined as output capable fields and can be
A* affected by the CLEAR operation.  Indicator 10 can also be
A* changed by the CLEAR operation even though it conditions an
A* input only field because field indicators are all treated
A* as output fields. The reason for this is that *ALL was not specifie
A* on the CLEAR operation
A*
A*N01N02N03T.Name++++++RLen++TDpBLinPosFunctions++++++++++++++++++++*
A          R FMT01
A  10        Field1        10A  I  2 30
A            Field2        10A  O  3 30
A            Field3        10A  B  4 30
A*
A*  End of DDS source
A*

F*Flename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
FWORKSTN   CF   E             WORKSTN INCLUDE(FMT01)
F
D*Name++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
D IN              C                    'INPUT DATA'

 /FREE
   CLEAR FMT01;
   WRITE FMT01;

   // Loop until PF03 is pressed
   DOW NOT *IN03;
      READ FMT01;
      *INLR = %EOF;

      // PF04 will transfer input fields to output fields.
      IF *IN04;
         Field2 = Field3;
         Field3 = Field1;
         CLEAR *IN04;
      ENDIF;
      Field1 = IN;

      // When PF11 is pressed, all the fields in the record format
      // defined as output or both will be reset to the values they
      // held after the initialization step.
      IF *IN11;
         RESET FMT01;
         CLEAR *IN11;
      ENDIF;

      // When PF12 is pressed, all the fields in the record
      // format defined as output or both will be cleared.
      IF *IN12;
         CLEAR FMT01;
         CLEAR *IN12;
      ENDIF;

      IF NOT *IN03;
         WRITE FMT01;
      ENDIF;
   ENDDO;

   *INLR = *ON;
 /END-FREE
```

Figure 298. Field Initialization for the CLEAR Record Format

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 explicit CLOSE operation closes one or more files or devices and disconnects them from the module. The file cannot be used again in the module unless you specify an explicit OPEN for that 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 files defined on global File specifications at once. Specifying CLOSE *ALL in a subprocedure does not have any effect on local files in the subprocedure. To close all the local files in a subprocedure, you must code a separate CLOSE operation for each file. 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 79.

Positions 71, 72, 75, and 76 must be blank.

If an array or table is to be written to an output file (specified using the TOFILE keyword) the array or table dump does not occur at LR time if the file is closed by a CLOSE operation). If the file is closed, it must be reopened for the dump to occur.

For more information, see "File Operations" on page 453.

```
*..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
     //  module. 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 299. CLOSE Operation

COMMIT (Commit)

Free-Form Syntax	COMMIT{(E)} {*boundary*}

Code	Factor 1	Factor 2	Result Field	Indicators		
COMMIT (E)	boundary			_	ER	_

The COMMIT operation:

- Makes all the changes to your files, opened for commitment control, that have been specified in output operations since the previous commit or rollback "ROLBK (Roll Back)" on page 798 operation (or since the beginning of operations under commitment control if there has been no previous commit or rollback operation). You specify a file to be opened for commit by specifying the COMMIT keyword on the file specification.
- Releases all the record locks for files you have under commitment control.

The file changes and the record-lock releases apply to all the files you have under commitment control, whether the changes have been requested by the program issuing the COMMIT operation, or by another program in the same activation group or job, dependent on the commit scope specified on the STRCMTCTL command. The program issuing the COMMIT operation does not need to have any files under commitment control. The COMMIT operation does not change the file position.

Commitment control starts when the CL command STRCMTCTL is executed. See the section on "Commitment Control" in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

For the *boundary* operand, , you can specify a constant or variable (of any type except pointer) to identify the boundary between the changes made by this COMMIT operation and subsequent changes. If *boundary* is not specified, the identifier is null.

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/Errors" on page 96.

For more information, see "File Operations" on page 453.

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 can 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. 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.

You must specify at least one resulting indicator in positions 71 through 76. Do not specify the same indicator for all three conditions. When specified, the resulting indicators are set on or off (for each cycle) to reflect the results of the compare.

For further rules for the COMP operation, see "Compare Operations" on page 445.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  Initial field values are:
 *                  FLDA = 100.00
 *                  FLDB = 105.00
 *                  FLDC = 100.00
 *                  FLDD = ABC
 *                  FLDE = ABCDE
 *
 *  Indicator 12 is set on; indicators 11 and 13 are set off.
C     FLDA          COMP      FLDB                                   111213
 *
 *  Indicator 15 is set on; indicator 14 is set off.
C     FLDA          COMP      FLDB                                   141515
 *
 *  Indicator 18 is set on; indicator 17 is set off.
C     FLDA          COMP      FLDC                                   171718
 *
 *  Indicator 21 is set on; indicators 20 and 22 are set off
C     FLDD          COMP      FLDE                                   202122
```

Figure 300. 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 operation code 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 result field pointer will not be changed if an error occurs, even if 'N' is specified. For more information on error handling, see “Program Exception/Errors” on page 96.

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.

When RPG memory management operations for the module are using single-level heap storage due to the ALLOC keyword on the Control specification, the DEALLOC operation can only handle pointers to single-level heap storage. When RPG memory management operations for the module are using teraspace heap storage, the DEALLOC operation can handle pointers to both single-level and teraspace heap storage.

For more information, see “Memory Management Operations” on page 458.

```
 *..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 301. 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			

Depending on the factor 1 entry, the declarative DEFINE operation can do either of the following:

- Define a field based on the attributes (length and decimal positions) of another field .
- Define a field as a data area .

You can specify the DEFINE operation anywhere within calculations, although you cannot specify a *DTAARA DEFINE in a subprocedure or use it with a UCS-2 result field. The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, the LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry is used for documentation only. Conditioning indicator entries (positions 9 through 11) are not permitted.

*LIKE DEFINE

The “DEFINE (Field Definition)” operation with *LIKE in factor 1 defines a field based upon the attributes (length and decimal positions) of another field.

Factor 2 must contain the name of the field being referenced, and the result field must contain the name of the field being defined. The field specified in factor 2, which can be defined in the program or externally, provides the attributes for the field being defined. Factor 2 cannot be a literal, 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 cannot be an array, an array element, a data structure, or a table name. Attributes such as ALTSEQ(*NO), NOOPT, ASCEND, CONST or null capability are not inherited from factor 2 by the result field. Only the data type, length, and decimal positions are inherited.

You can use positions 64 through 68 (field length) to make the result field entry longer or shorter than the factor 2 entry. A plus sign (+) preceding the number indicates a length increase; a minus sign (-) indicates a length decrease. Positions 65-68 can contain the increase or decrease in length (right-adjusted) or can be blank. If positions 64 through 68 are blank, the result field entry is defined with the same length as the factor 2 entry. You cannot change the number of decimal positions for the field being defined. 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 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 of the module. 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.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
*
 *  FLDA is a 7-position character field.
 *  FLDB is a 5-digit field with 2 decimal positions.
 *
 *
 *  FLDP is a 7-position character field.
C     *LIKE         DEFINE    FLDA          FLDP
 *
 *  FLDQ is a 9-position character field.
C     *LIKE         DEFINE    FLDA          FLDQ             +2
 *
 *  FLDR is a 6-position character field.
C     *LIKE         DEFINE    FLDA          FLDR            - 1
 *
 *  FLDS is a 5-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDS
 *
 *  FLDT is a 6-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDT           +  1
 *
 *  FLDU is a 3-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDB          FLDU            - 2
 *
 *  FLDX is a 3-position numeric field with 2 decimal positions.
C     *LIKE         DEFINE    FLDU          FLDX
```

*Figure 302. DEFINE Operation with *LIKE*

Note the following for *LIKE DEFINE of numeric fields:

- If the field is fully defined on Definition Specifications, the format is not changed by the *LIKE DEFINE.
- Otherwise, if the field is a subfield of a data structure, it is defined in zoned format.
- Otherwise, the field is defined in packed format.

```
D            DS
D  Fld1
D  Fld2      S              7P 2
 *
 * Fld1 will be defined as zoned because it is a subfield of a
 * data structure and numeric subfields default to zoned format.
 *
C    *LIKE      DEFINE      Fld2       Fld1
 *
 * Fld3 will be defined as packed because it is a standalone field
 * and all numeric items except subfields default to packed format.
C    *LIKE      DEFINE      Fld1       Fld3
```

*Figure 303. Using *LIKE DEFINE*

*DTAARA DEFINE

The “DEFINE (Field Definition)” on page 651 operation with *DTAARA in factor 1 associates a field, a data structure, a data-structure subfield, or a data-area data structure (within your ILE RPG program) with an AS/400 data area (outside your ILE RPG program).

Note: You cannot use *DTAARA DEFINE within a subprocedure or with a UCS-2 result field.

In factor 2, specify the external name of a data area. Use *LDA for the name of the local data area or use *PDA for the Program Initialization Parameters (PIP) data area. If you leave factor 2 blank, the result field entry is both the RPG IV name and the external name of the data area.

In the result field, specify the name of one of the following that you have defined in your program: a field, a data structure, a data structure subfield, or a data-area data structure. You use this name with the IN and OUT operations to retrieve data from and write data to the data area specified in factor 2. When you specify a data-area data structure in the result field, the ILE RPG program implicitly retrieves data from the data area at program start and writes data to the data area when the program ends.

The result field entry must not be the name of a program-status data structure, a file-information data structure (INFDS), a multiple-occurrence data structure, an input record field, an array, an array element, or a table. It cannot be the name of a subfield of a multiple-occurrence data structure, of a data area data structure, of a program-status data structure, of a file-information data structure (INFDS), or of a data structure that already appears on a *DTAARA DEFINE statement, or has already been defined as a data area using the DTAARA keyword on a definition specification.

You can create three kinds of data areas:

- *CHAR Character
- *DEC Numeric
- *LGL Logical

You can also create a DDM data area (type *DDM) that points to a data area on a remote system of one of the three types above.

Only character and numeric types (excluding float numeric) are allowed to be associated with data areas. The actual data area on the system must be of the same type as the field in the program, with the same length and decimal positions. Indicator fields can be associated with either a logical or character data area.

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.

In positions 64 through 70, you can define the length and number of decimal positions for the entry in the result field. These specifications must match those for the external description of the data area specified in factor 2. The local data area is character data of length 1024, but within your program you can access the local data area as if it has a length of 1024 or less.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The attributes (length and decimal positions) of
 *  the data area (TOTGRS) must be the same as those for the
 *  external data area.
C
C     *DTAARA       DEFINE                  TOTGRS           10 2
C
 *
 *  The result field entry (TOTNET) is the name of the data area to
 *  be used within the ILE RPG program.  The factor 2 entry (TOTAL)
 *  is the name of the data area as defined to the system.
C
C     *DTAARA       DEFINE    TOTAL         TOTNET
C
 *
 *  The result field entry (SAVTOT) is the name of the data area to
 *  be used within the ILE RPG program.  The factor 2 entry (*LDA)
 *  indicates the use of the local data area.
C
C     *DTAARA       DEFINE    *LDA          SAVTOT
```

*Figure 304. DEFINE Operation with *DTAARA*

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 or record format)		NR	ER	_

The DELETE operation deletes a record from a database file. The file must be an update file (identified by a U in position 17 of the file description specifications) The deleted record can never be retrieved.

If a search argument (*search-arg*) is not specified, the DELETE operation deletes the current record (the last record retrieved). The record must have been locked by a previous input operation (for example, CHAIN or READ).

The search argument, *search-arg*, must be the key or relative record number used to retrieve the record to be deleted. If access is by key, *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 546. If access is by relative record number, *search-arg* must be an integer literal or a numeric field with zero decimal positions. For an example of using a list of values to search for the record to be deleted, see Figure 289 on page 635.

The *name* operand must be the name of the update file or a record format in the file from which a record is to be deleted. A record format name is valid only with an externally described file. If *search-arg* is not specified, the record format name must be the name of the last record read from the file; otherwise, an error occurs.

If *search-arg* 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 *search-arg* 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 79.

Under the IBM i operating system, if a read operation is done on the file specified in *file-name* after a successful DELETE operation to that file, the next record after the deleted record is obtained.

See “Database Null Value Support” on page 219 for information on handling records with null-capable fields and keys.

For more information, see “File Operations” on page 453.

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 (result) is placed in the result field. If factor 1 is 0, the result of the divide operation is 0. Factor 2 cannot be 0. If it is, an error occurs and the RPG IVexception/error handling routine receives control. When factor 1 is not specified, the result field (dividend) is divided by factor 2 (divisor), and the result (quotient) is placed in the result field. 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.

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, you must ensure that the DIV operation is processed immediately before the MVR operation. If the MVR operation is processed before the DIV operation, undesirable results occur. If move remainder is the next operation, the result of the divide operation cannot be half-adjusted (rounded).

For further rules for the DIV operation, see “Arithmetic Operations” on page 434.

Figure 172 on page 437 shows examples of 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.

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 further information on DO groups, see “Structured Programming Operations” on page 469.

In factor 1, specify a starting value with zero decimal positions, using a numeric literal, named constant, or field name. If you do not specify factor 1, the starting value is 1.

In factor 2, specify the limit value with zero decimal positions, using a numeric field name, literal, or named constant. If you do not specify factor 2, the limit value is 1.

In the result field, specify a numeric field name that will contain the current index value. The result field must be large enough to contain the limit value plus the increment. If you do not specify an index field, one is generated for internal use. Any value in the index 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. It can be a numeric literal or a numeric field with no decimal positions. If it is blank, the value to be added to the index field is 1.

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).

Remember the following when specifying the DO operation:
- The index, increment, limit value, and indicators can be modified within the loop to affect the ending of the DO group.
- A DO group cannot span both detail and total calculations.

See “LEAVE (Leave a Do/For Group)” on page 708 and “ITER (Iterate)” on page 703 for information on how those operations affect a DO operation.

See “FOR (For)” on page 692 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 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The DO group is processed 10 times when indicator 17 is on;
 *  it stops running when the index value in field X, the result
 *  field, is greater than the limit value (10) in factor 2.  When
 *  the DO group stops running, control passes to the operation
 *  immediately following the ENDDO operation.  Because factor 1
 *  in the DO operation is not specified, the starting value is 1.
 *  Because factor 2 of the ENDDO operation is not specified, the
 *  incrementing value is 1.
C
C   17              DO        10            X                 3 0
C                   :
C                   ENDDO
 *
 *  The DO group can be processed 10 times.  The DO group stops
 *  running when the index value in field X is greater than
 *  the limit value (20) in factor 2, or if indicator 50 is not on
 *  when the ENDDO operation is encountered.  When indicator 50
 *  is not on, the ENDDO operation is not processed; therefore,
 *  control passes to the operation following the ENDDO operation.
 *  The starting value of 2 is specified in factor 1 of the DO
 *  operation, and the incrementing value of 2 is specified in
 *  factor 2 of the ENDDO operation.
 *
C     2             DO        20            X                 3 0
C                   :
C                   :
C                   :
C   50              ENDDO     2
```

Figure 305. 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 code precedes a group of operations which you want to execute at least once and possibly more than once. Its function is similar to that of the DOUxx operation code. 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 486.

For fixed-format syntax, level and conditioning indicators are valid. Factor 1 must be blank. Extended factor 2 contains the expression to be evaluated.

For more information, see "Compare Operations" on page 445 or "Structured Programming Operations" on page 469.

```
*..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 306. 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 further information on DO groups and the meaning of xx, see "Structured Programming Operations" on page 469.

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 DOUxx group are processed once, and then the group is repeated until either:

- the relationship exists between factor 1 and factor 2
- 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 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 DOUxx 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 708 and "ITER (Iterate)" on page 703 for information on how those operations affect a DOUxx operation.

For more information, see "Compare Operations" on page 445 or "Structured Programming Operations" on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The DOUEQ operation runs the operation within the DO group at
 *  least once.
C
C     FLDA          DOUEQ     FLDB
C
 *
 *  At the ENDDO operation, a test is processed to determine whether
 *  FLDA is equal to FLDB.  If FLDA does not equal FLDB, the
 *  preceding operations are processed again.  This loop continues
 *  processing until FLDA is equal to FLDB.  When FLDA is equal to
 *  FLDB, the program branches to the operation immediately
 *  following the ENDDO operation.
C
C                   SUB       1             FLDA
C                   ENDDO
C
 *
 *  The combined DOUEQ ANDEQ OREQ operation processes the operation
 *  within the DO group at least once.
C
C     FLDA          DOUEQ     FLDB
C     FLDC          ANDEQ     FLDD
C     FLDE          OREQ      100
C
 *
 *  At the ENDDO operation, a test is processed to determine whether
 *  the specified condition, FLDA equal to FLDB and FLDC equal to
 *  FLDD, exists.  If the condition exists, the program branches to
 *  the operation immediately following the ENDDO operation.  There
 *  is no need to test the OREQ condition, FLDE equal to 100, if the
 *  DOUEQ and ANDEQ conditions are met.  If the specified condition
 *  does not exist, the OREQ condition is tested.  If the OREQ
 *  condition is met, the program branches to the operation
 *  immediately following the ENDDO.  Otherwise, the operations
 *  following the OREQ operation are processed and then the program
 *  processes the conditional tests starting at the second DOUEQ
 *  operation.  If neither the DOUEQ and ANDEQ condition nor the
 *  OREQ condition is met, the operations following the OREQ
 *  operation are processed again.
C
C                   SUB       1             FLDA
C                   ADD       1             FLDC
C                   ADD       5             FLDE
C                   ENDDO
```

Figure 307. 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. Its function is similar to that of the DOWxx operation code. 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 DOW operation are performed while the expression in *indicator-expression* is true. See Chapter 20, “Expressions,” on page 477 for details on expressions. For information on how operation extenders M and R are used, see “Precision Rules for Numeric Operations” on page 486.

For fixed-format syntax, level and conditioning indicators are valid. Factor 1 must be blank. Factor 2 contains the expression to be evaluated.

For more information, see “Compare Operations” on page 445 or “Structured Programming Operations” on page 469.

```
*..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 308. 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 469.

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 445.

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 708 and “ITER (Iterate)” on page 703 for information on how those operations affect a DOWxx operation.

For more information, see “Compare Operations” on page 445 or “Structured Programming Operations” on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The DOWLT operation allows the operation within the DO group
 *  to be processed only if FLDA is less than FLDB.  If FLDA is
 *  not less than FLDB, the program branches to the operation
 *  immediately following the ENDDO operation.  If FLDA is less
 *  than FLDB, the operation within the DO group is processed.
C
C     FLDA          DOWLT     FLDB
C
 *
 *  The ENDDO operation causes the program to branch to the first
 *  DOWLT operation where a test is made to determine whether FLDA
 *  is less than FLDB.  This loop continues processing until FLDA
 *  is equal to or greater than FLDB; then the program branches
 *  to the operation immediately following the ENDDO operation.
C
C                   MULT      2.08          FLDA
C                   ENDDO
C
 *  In this example, multiple conditions are tested.  The combined
 *  DOWLT ORLT operation allows the operation within the DO group
 *  to be processed only while FLDA is less than FLDB or FLDC. If
 *  neither specified condition exists, the program branches to
 *  the operation immediately following the ENDDO operation. If
 *  either of the specified conditions exists, the operation after
 *  the ORLT operation is processed.
C
C     FLDA          DOWLT     FLDB
C     FLDA          ORLT      FLDC
C
 *  The ENDDO operation causes the program to branch to the second
 *  DOWLT operation where a test determines whether specified
 *  conditions exist.  This loop continues until FLDA is equal to
 *  or greater than FLDB and FLDC; then the program branches to the
 *  operation immediately following the ENDDO operation.
C
C                   MULT      2.08          FLDA
C                   ENDDO
```

Figure 309. DOWxx Operations

DSPLY (Display Message)

Free-Form Syntax	DSPLY{(E)} {*message* {*message-queue* {*response*}}}

Code	Factor 1	Factor 2	Result Field	Indicators		
DSPLY (E)	message	message-queue	response	_	ER	_

The DSPLY operation allows the program to communicate with the display work station that requested the program. Either *message*, *response*, or both operands must be specified. The operation can display a message and accept a response.

The value in the *message* operand and possibly the *response* operand are used to create the message to be displayed. *message* can be a field name, a literal, a named constant, a table name, or an array element whose value is used to create the message to be displayed. Within free-form calculations, the message operand can be an expression, provided the expression is enclosed by parentheses. The *message* operand can also be *M, followed by a message identifier that identifies the message to be retrieved from the message file, QUSERMSG. Use the OVRMSGF CL command to use a different message file. QUSERMSG must be in a library in the library list of the job receiving the message.

The message identifier must be 7 characters in length consisting 3 alphabetic characters and four numeric characters (for example, *MUSR0001, this means message USR0001 is used).

If specified, the *message-queue* operand can be a character field, a literal, a named constant, a table name, or an array element whose value is the symbolic name of the object meant to receive the message and from which the optional response can be sent. Any queue name, except a program message queue name, can be the value contained in the *message-queue* operand. The queue must be declared to the operating system before it can be used during program execution. (For information on how to create a queue, see the *CL Programming*). There are two predefined queues:

Queue **Value**

QSYSOPR
The message is sent to the system operator. Note that the QSYSOPR message queue severity level must be zero (00) to enable the DSPLY operation to immediately display a message to the system operator.

***EXT**
The message is sent to the external message queue.

Note: For a batch job, if no *message-queue* value is specified, the default is QSYSOPR. For an interactive job, the default value is *EXT.

The *response* operand is optional. If it is specified, the response is placed in it. *response* can be a field name, a table name, or an array element in which the response is placed. If no data is entered, *response* is unchanged. To specify a response but no message queue in a free-form specification, specify `' '` for *message-queue*.

Fully qualified names may be specified as the Result-Field operand, and expressions are allowed as Factor 1 and Factor 2 operands, when 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.

To handle DSPLY exceptions (program status code 333), 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/Errors” on page 96.

When you specify the DSPLY operation *with no message identifier in the message operand*, the operation functions as follows:

- If the *message* operand is specified but the *response* operand is not, the contents of the *message* operand are displayed. The program does not wait for a response unless a display file with the parameter RSTDSP (*NO) specified was used to display a format at the workstation. Then the program waits for the user to press Enter.
- If the *message* operand is not specified but the *response* operand is, the contents of the *response* operand are displayed and the program waits for the user to enter data for the response. The reply is placed in the *response* operand.
- When both *message* and *response* operands are specified,, their contents are combined and displayed. The program waits for the user to enter data for the response. The response is placed in the result field.
- If you request help on the message, you can find the type and attributes of the data that is expected and the number of unsuccessful attempts that have been made.

  The maximum length of information that can be displayed is 52 bytes.

The format of the record written by the DSPLY operation with no message identifier specified by the *message* operand follows:

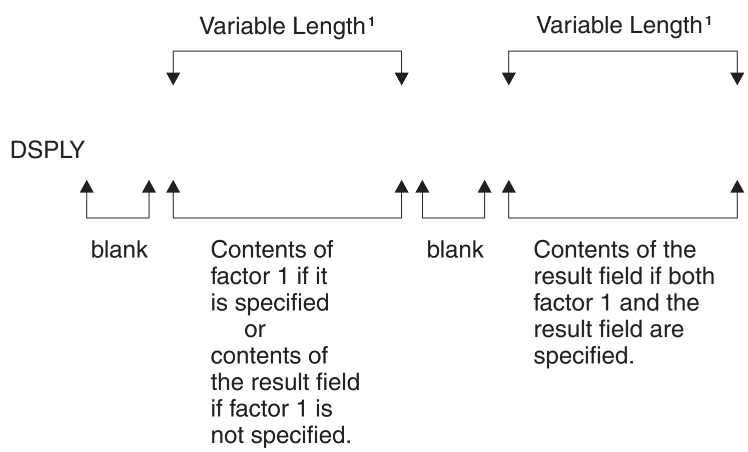


Figure 310. DSPLY Operation Record Format. [1]The maximum length of information that can be displayed is 52 bytes.

When you specify the DSPLY operation *with a message identifier in the message operand*, the operation functions as follows: the message identified in the *message* operand is retrieved from QUSERMSG, the message is displayed, and the program waits for the user to respond by entering data if the *response* operand is specified. The response is placed in the result field.

When replying to a message, remember the following:

- Non-float numeric fields sent to the display are right-adjusted and zero-suppressed.
- If a non-float numeric field is entered with a length greater than the number of digits in the result field and the rightmost character is not a minus sign (-), an error is detected and a second wait occurs. The user must key in the field again.
- A float value is entered in the external display representation. It can be no longer than 14 characters for 4-byte float fields, and no longer than 23 characters for 8-byte float fields.
- If graphic, UCS-2, or character data is entered, the length must be equal or less than the receiving field length.
- If the result field is variable-length, its length will be set to the length of the value that you enter.
- If a date, time, or timestamp field is entered, the format and separator must match the format and separator of the result field. If the format or separator do not match, or the value is not valid (for example a date of 1999/99/99), an error is detected and a second wait occurs. The user must key in the field again.
- The DSPLY operation allows the workstation user up to 5 attempts to respond to the message. After the fifth unsuccessful attemp, the DSPLY operation fails. If the DSPLY operation does not have a message identifier specified in the *message* operand, the user can request help on the message to find the type and attributes of the expected response.
- To enter a null response to the system operator queue (QSYSOPR), the user must enter the characters *N and then press Enter.
- Graphic, UCS-2, or character fields are padded on the right with blanks after all characters are entered.
- UCS-2 fields are displayed and entered as single-byte characters.
- Numeric fields are right-adjusted and padded on the left with zeros after all characters are entered.
- Lowercase characters are not converted to uppercase.
- If factor 1 or the result field is of graphic data type, they will be bracketed by SO/SI when displayed. The SO/SI will be stripped from the value to be assigned to the graphic result field on input.
- Float fields are displayed in the external display representation. Float values can be entered as numeric literals or float literals. When entering a response, the float value does not have to be normalized.

For more information, see “Message Operation” on page 460.

```
/free
    // Display prompt and wait for response:
    dsply prompt '' result;
    // Display string constructed in an expression:
    dsply ('Length of name is ' + %char(%len(str)) + ' bytes.');
/end-free
```

Figure 311. DSPLY Operation Code Examples

DUMP (Program Dump)

Free-Form Syntax	DUMP{(A)} {*identifier*}

Code	Factor 1	Factor 2	Result Field	Indicators		
DUMP (A)	identifier					

The DUMP operation provides a dump (all fields, all files, indicators, data
structures, arrays, and tables defined) of the module. It can be used independently
or in combination with the IBM i testing and debugging functions. When the
OPTIMIZE(*FULL) compiler option is selected on either the CRTBNDRPG or
CRTRPGMOD command or as a keyword on a control specification, the field
values shown in the dump may not reflect the actual content due to the effects of
optimization.

If the DBGVIEW(*NONE) compiler option is specified, the dump will only show the program status data structure, the file information data structures, and the *IN indicators. Other variables will not have their contents shown because the object does not contain the necessary observability information.

If the DEBUG(*NO) control-specification keyword is specified, no dump is performed. You can override this keyword by specifying operation extender A. This operation extender means that a dump is always performed, regardless of the value of the DEBUG keyword.

The contents of the optional *identifier* operand identify the DUMP operation. It will replace the default heading on the dump listing if specified. It must contain a character or graphic entry that can be one of: a field name, literal, named constant, table name, or array element whose contents identify the dump. If the *identifier* operand is a graphic entry, it is limited to 64 double byte characters. *identifier* cannot be a figurative constant.

The program continues processing the next calculation statement following the DUMP operation.

The DUMP operation is performed if the DEBUG keyword is specified on the control specification, or the A operation extender is coded for the DUMP operation. Otherwise, the DUMP operation is checked for errors and the statement is printed on the listing, but the DUMP operation is not processed.

When dumping files, the DUMP will dump the File Feedback Information section of the INFDS, but not the Open Feedback Information or the Input/Output Feedback Information sections of the INFDS. DUMP will instead dump the actual Open Feedback, and Device Feedback Information for the file.

Note that if the INFDS you have declared is not large enough to contain the Open Feedback, or Input/Output Feedback Information, then you do not have to worry about doing a POST before DUMP since the File Feedback Information in the INFDS is always up to date.

The values of variables in subprocedures may not be valid if the subprocedure is not active. If a subprocedure has been called recursively, the values from the most recent invocation are shown.

Java object variables may not show the expected value. The RPG module may retain the reference to an object after the object no longer exists; it is possible for an object reference to be reused, and refer to a different object that is unrelated to the RPG module being dumped. That different object is the one that will appear in the formatted dump.

For an sample dump listing, see the chapter on obtaining dumps in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

For more information, see "Information Operations" on page 457.

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.

Within total calculations, the control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry is for documentation purposes only. Conditioning indicator entries (positions 9 through 11) are not permitted. To close the IFxx/ELSE group use an ENDIF operation.

Figure 326 on page 700 shows an example of an ELSE operation with an IFxx operation.

For more information, see “Structured Programming Operations” on page 469.

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 486.

For more information, see “Structured Programming Operations” on page 469.

```
*..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
/free

   IF keyPressed = HELPKEY;
      displayHelp();
   ELSEIF keyPressed = EXITKEY;
      return;
   ELSEIF keyPressed = ROLLUP OR keyPressed = ROLLDOWN;
      scroll (keyPressed);
   ELSE;
      signalError ('Key not defined');
   ENDIF;

/end-free
```

Figure 312. 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 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

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 one of: an array element, table name, data structure, field, named constant, or numeric literal. If *increment-value* is not specified on the ENDDO, the increment defaults to 1. If *increment-value* is negative, the DO group will never end.

Conditioning indicators are optional for ENDDO or ENDFOR and not allowed for ENDCS, ENDIF, ENDMON, and ENDSL.

Resulting indicators are not allowed. No operands are allowed for ENDCS, ENDIF, ENDMON, and 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.

See the CASxx, DO, DOUxx, DOWxx, FOR, IFxx, and DOU, DOW, IF, MONITOR, and SELECT operations for examples that use the ENDyy operation.

For more information, see "Error-Handling Operations" on page 452 or "Structured Programming Operations" on page 469.

ENDSR (End of 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 an RPG IV subroutine and the return point (*return-point*) to the cycle-main program. ENDSR must be the last statement in the subroutine. In traditional 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 control level entry (positions 7 and 8) can be SR or blank. Conditioning indicator entries are not allowed.

The ENDSR operation ends a subroutine and causes a branch back to the statement immediately following the EXSR or CASxx operation unless the subroutine is a program exception/error subroutine (*PSSR) or a file exception/error subroutine (INFSR). For these subroutines, the *return-point* operand of the ENDSR operation can contain an entry that specifies where control is to be returned following processing of the subroutine. This entry can be a field name that contains a reserved keyword or a literal or named constant that is a reserved keyword. If a return point that is not valid is specified, the RPG IV error handler receives control.

Note: The *return-point* operand cannot be specified for an ENDSR operation that occurs within a subprocedure (including a linear-main procedure).

See "File Exception/Error Subroutine (INFSR)" on page 93 for more detail on return points.

See Figure 183 on page 474 for an example of coding an RPG IV subroutine.

For more information, see "Subroutine Operations" on page 472.

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 `"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 171. 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, and if the variable does not have the same name as an operation code.

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 20, "Expressions," on page 477 for general information on expressions. See "Precision Rules for Numeric Operations" on page 486 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 313. 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.

Note: 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.

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 171. Otherwise, the expression is evaluated once and the value is placed into each element of the array or sub-array.

See Chapter 20, “Expressions,” on page 477 for general information on expressions. See “Precision Rules for Numeric Operations” on page 486 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 314. EVALR Operations

EVAL-CORR (Assign corresponding subfields)

Free-Form Syntax	EVAL-CORR{(HMR)} *target = source*;

Code	Factor 1	Extended Factor 2
EVAL-CORR		*target = source*

The EVAL-CORR operation assigns data and null-indicators from the corresponding subfields of the source data structure to the subfields of the target data structure. The subfields that are assigned are the subfields that have the same name and compatible data type in both data structures. For example, if data structure DS1 has character subfields A, B, and C, and data structure DS2 has character subfields B, C, and D, statement

```
EVAL-CORR DS1 = DS2
```

will assign data from subfields DS2.B and DS2.C to DS1.B and DS1.C. Null-capable subfields in the target data structure that are affected by the EVAL-CORR operation will also have their null-indicators set from the null-indicator from the source data structure's subfield, or to *OFF, if the source subfield is not null-capable.

If an operation code extender H is specified, the half-adjust function applies on all numeric assignments. Extenders for EVAL-CORR can be specified only in Free-form calculations.

If operation code extender M or R is specified, it applies to the arguments of any procedure call specified as part of the source or target expression. Extenders for EVAL-CORR can be specified only in Free-form calculations.

The *EVAL-CORR Summary section* in the compiler listing can be used to determine
- which subfields were selected to be affected by the EVAL-CORR operation
- for subfields not selected, the reason the subfield was not selected
- for subfields that are selected, any additional information about the subfields such as a difference in the dimension or null-capability of the subfields.

See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information about the EVAL-CORR Summary section.

Remember the following when using the EVAL-CORR operation:
- Operation code EVAL-CORR may be coded either in free-form calculations or in fixed-form calculations. When coded in fixed-form calculations, the assignment expression is coded in the Extended Factor 2 entry, with the Factor 1 entry left blank.
- The source and target operands must both be data structure variables, including data structure subfields defined with LIKEDS or LIKEREC.
- The operands may be qualified or unqualified data structures. However, for the operation to be successful, at least one of the operands must be a qualified data structure; otherwise, it would not be possible for the two data structures to have any subfields with the same name.
- The subfields involved in the assignment are those that have the same name in both data structures and have data types that are compatible for assignment using EVAL.
- When comparing the subfield names to find corresponding subfieds, the names used are the internal program names; the internal program names may be different from the external names in the case of fields from externally-described

files or data structures. For fields defined externally and renamed or prefixed, the name used is the name after applying the rename or prefix.

- For subfields in the source and target that correspond by name and are both data structures defined with LIKEDS or LIKEREC, the subfields that are assigned are the corresponding subfields of the subfield data structures. If two subfields in the source and target have the same name but one is a data structure defined with LIKEDS or LIKEREC, and the other is not a data structure, the subfield is not assigned by the EVAL-CORR operation.
- The assignment of data from the source subfields to the target subfields follows the same rules as for operation code EVAL. For example, character values are assigned left adjusted with truncation or padding with blanks for unequal lengths.
- Data is assigned subfield by subfield by the order of subfields in the source data structure. If there are overlapping subfields in the target data structure, either due to overlapping from-and-to positions or due to the OVERLAY keyword, later assignment may overwrite earlier moves.
- When the source and target data structures or corresponding source and target subfields which are both data structures are defined the same way with LIKEDS or LIKEREC, that is, both data structures are defined like the same data structure, the compiler will optimize the assignment and assign the data structure as a whole, and not as a series of individual subfield assignments.
- If either the source or target operand is a multiple occurrence data structure, the current occurrence is used.
- If you are working with arrays:
  – If the source operand is an unindexed array data structure, the target data structure must also be an array data structure.
  – If the target operand is an unindexed array data structure, the operation works on each element of the array data structure, following the same rules as EVAL with an array result. %SUBARR may be used to restrict the number of elements used in either the source or target data structure array.
  – If one subfield is an array, both subfields must be arrays. If the dimension of one array subfield is smaller than the other, only the smaller number of array elements is assigned. If the target subfield has more elements, the additional elements are unchanged by the EVAL-CORR operation.
- If you are working with null-capable subfields:
  – EVAL-CORR automatically handles assignment of null-indicators for null-capable subfields that are not data structure subfields.
    - If both the source and target subfields are null-capable, the source subfield's null-indicator is copied to the target subfield's null-indicator.
    - If the target subfield is null-capable and the source subfield is not null-capable, the target subfield's null-indicator is set to *OFF.
    - If the source subfield is null-capable and the target subfield is not null-capable, the source subfield's null-indicator is ignored.
    - The EVAL-CORR operation sets the null-indicators for scalar and array subfields only. If a null-capable subfield is a data structure, its null-indicator will not be set by the EVAL-CORR operation; similarly, if the target data structure itself is null-capable, its null-indicator will not be set by the EVAL-CORR operation..
  – If the subfield is a data structure and a null-indicator is assigned to the data structure itself, the null-indicator is not affected by the EVAL-CORR operation.

Examples of the EVAL-CORR operation

```
 * Physical file EVALCORRPF
A          R PFREC
A            NAME          25A
A            IDNO          10P 0
A            CITY          20A
 * Display file EVALCORRDF
A          R DSPFREC
A                                   3  2'Name'
A            NAME          25A  O  3 15
A                                   4  2'City'
A            CITY          20A  B  4 15CHECK(LC)
 * RPG program
Fevalcorrpfuf   e             disk
Fevalcorrdfcf   e             workstn

D pf_ds         e ds                  extname(evalcorrpf : *input)
D                                     qualified
D pf_save_ds      ds                  likeds(pf_ds)
D dspf_ds       e ds                  extname(evalcorrdf : *all)
D                                     qualified
 /free
       read pfrec pf_ds;
       dow not %eof;
          // Assign all subfields with the same name and type
          // to the data structure for the EXFMT operation
          // to the display file (NAME and CITY)
          eval-corr dspf_ds = pf_ds;

          // Show the screen to the user
          exfmt dspfrec dspf_ds;
          // Save the original physical file record
          // and assign the display file subfields to the
          // physical file data structure.  Then compare
          // the physical file data structure to the saved
          // version to see if any fields have changed.
          eval pf_save_ds = pf_ds;
          eval-corr pf_ds = dspf_ds;
          if pf_ds <> pf_save_ds;
             // Some of the fields have changed
             update pfrec pf_ds;
          endif;
          read pfrec pf_ds;
       enddo;
       *inlr = '1';
```

Figure 315. EVAL-CORR with externally-described data structure I/O

```
 * The two data structures ds1 and ds2 have several
 * subfields, some having the same names and
 * compatible types:
 *    num     - appears in both, has compatible type
 *    extra   - appears only in ds1
 *    char    - appears in both, has identical type
 *    other   - appears only in ds1
 *    diff    - appears in both, types are not compatible
 *    another - appears only in ds2
D ds1             ds                  qualified
D   num                         10i 0
D   extra                         d
D   char                        20a
D   otherfld                     1a
D   diff                         5p 0
D ds2             ds                  qualified
D   char                        20a
D   diff                         5a
D   another                      5a
D   num                         15p 5

 /free
       // assign corresponding fields from DS1 to DS2
       EVAL-CORR ds2 = ds1;
       // this EVAL-CORR is equivalent to these EVAL operations
       // between all the subfields which have the same name
       // in both data structures and which have a compatible
       // data type
       //   EVAL ds2.num = ds1.num;
       //   EVAL ds2.char = ds1.char;
       // - Subfields "extra" and "another" are not affected
       //   because there is no subfield of the same name in
       //   the other data structure.
       // - Subfield "diff" is not selected because the
       //   subfields do not a compatible type
```

Figure 316. EVAL-CORR between program-described data structures

```
 * DDS for file EVALCORRN1
A          R REC1
A            FLD1          10A         ALWNULL
A            FLD2          10A         ALWNULL
A            FLD3          10A
A            FLD4          10A
A            FLD5           5P 0       ALWNULL
 * DDS for file EVALCORRN2
A          R REC2
A            FLD1          10A         ALWNULL
A            FLD2          10A
A            FLD3          10A         ALWNULL
A            FLD4          10A
A            FLD5           5A         ALWNULL
 * In the following example, data structure "ds1"
 * is defined from REC1 in file EVALCORRN1 and
 * data structure "ds2" is defined from REC2 in
 * file EVALCORRN2 above.  The EVAL-CORR operation
 * does the following:
 * 1. DS2.FLD1 is assigned the value of DS1.FLD1
 *    and %NULLIND(DS2.FLD1) is assigned the value of
 *    %NULLIND(DS1.FLD1)
 * 2. DS2.FLD2 is assigned the value of DS1.FLD2
 * 3. DS2.FLD3 is assigned the value of DS1.FLD3
 *    and %NULLIND(DS2.FLD3) is assigned *OFF
 * 4. DS2.FLD4 is assigned the value of DS1.FLD4
 * The null-indicator for DS1.FLD2 is ignored because
 * the target subfield DS2.FLD2 is not null-capable.
 * DS2.FLD5 is ignored because DS1.FLD5 has a different
 * data type, so the subfields do not correspond.
H ALWNULL(*USRCTL)
FEVALCORRN1IF   E             DISK
FEVALCORRN2O    E             DISK
D ds1             DS                  LIKEREC(REC1:*INPUT)
D ds2             DS                  LIKEREC(REC2:*OUTPUT)
C                   READ      REC1          ds1
C                   EVAL-CORR ds2 = ds1
C                   WRITE     REC2          ds2
```

Figure 317. EVAL-CORR with null-capable subfields

```
D ds0             ds                   qualified
D   num                          10i 0
D   char                         20a   varying
 * A data structure with a nested subfield data structure
D ds1             ds                   qualified
D   a                                  likeds(ds0)
D   b                                  likeds(ds0)
D   char                         20a   varying
D   otherfld                      1a
 * A data structure with a nested subfield data structure
D ds2             ds                   qualified
D   char                         20a
D   another                       5a
D   b                                  likeds(ds0)

 /free
       // assign corresponding fields from DS1 to DS2
       EVAL-CORR ds2 = ds1;
       // this EVAL-CORR is equivalent to these EVAL operations
       //   EVAL ds2.b.num = ds1.b.num;
       //   EVAL ds2.b.char = ds1.b.char;
       //   EVAL ds2.char = ds1.char;
       // assign corresponding fields from DS1.A to DS0
       EVAL-CORR(H) ds0 = ds1.a;
       // this EVAL-CORR is equivalent to these EVAL operations
       //   EVAL(H) ds0.num = ds1.a.num;
       //   EVAL ds0.char = ds1.a.char;
       // assign corresponding fields from DS1.A to DS2.B
       EVAL-CORR ds2.b = ds1.a;
       // this EVAL-CORR is equivalent to these EVAL operations
       //   EVAL ds2.b.num = ds1.a.num;
       //   EVAL ds2.b.char = ds1.a.char;
```

Figure 318. EVAL-CORR with nested subfield data structures

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 records to be written during either detail calculations or total calculations. See Figure 319 on page 685 for examples of the EXCEPT operation.

When specifying the EXCEPT operation remember:

- The exception records that are to be written during calculation time are indicated by an E in position 17 of the output specifications. An EXCEPT name, which is the same name as specified by the *except-name* operand of an EXCEPT operation, can be specified in positions 30 through 39 of the output specifications of the exception records.
- Only exception records, not heading, detail, or total records, can contain an EXCEPT name.
- When the EXCEPT operation with a name specified in the *except-name* operand is processed, 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 with no name in positions 30 through 39 of the output specifications are checked and written if the conditioning indicators are satisfied.
- If an exception record is conditioned by an overflow indicator on the output specification, the record is written only during the overflow portion of the RPG IV cycle or during fetch overflow. The record is not written at the time the EXCEPT operation is processed.
- 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 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  When the EXCEPT operation with HDG specified in factor 2 is
 *  processed, all exception records with the EXCEPT name HDG are
 *  written.  In this example, UDATE and PAGE would be printed
 *  and then the printer would space 2 lines.
 *  The second HDG record would print a line of dots and then the
 *  printer would space 3 lines.
 *
C                   EXCEPT    HDG
 *
 *  When the EXCEPT operation with no entry in factor 2 is
 *  processed, all exception records that do not have an EXCEPT
 *  name specified in positions 30 through 39 are written if the
 *  conditioning indicators are satisfied.  Any exception records
 *  without conditioning indicators and without an EXCEPT name
 *  are always written by an EXCEPT operation with no entry in
 *  factor 2.  In this example,  if indicator 10 is on, TITLE and
 *  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
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 319. EXCEPT Operation with/without Factor 2 Specified

EXFMT (Write/Then Read Format)

Free-Form Syntax	EXFMT{(E)} *format-name {data-structure}*

Code	Factor 1	Factor 2	Result Field	Indicators		
EXFMT (E)		format-name	data-structure	_	ER	_

The EXFMT operation is a combination of a WRITE followed by a READ to the same record format. EXFMT is valid only for a WORKSTN file defined as a full procedural (F in position 18 of the file description specifications) combined file (C in position 17 of the file description specifications) that is externally described (E in position 22 of the file description specifications)

The *format-name* operand must be the name of the record format to be written and then read.

If the data-structure operand is specified, the record is written from and read into the data structure. The data structure must be a data structure defined with EXTNAME(...:*ALL) or LIKEREC(...:*ALL). See "File Operations" on page 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

To handle EXFMT exceptions (file status codes greater than 1000), either the operation code extender 'E' or an error indicator ER can be specified, but not both. When an error occurs, the read portion of the operation is not processed (record-identifying indicators and fields are not modified). For more information on error handling, see "File Exception/Errors" on page 79.

Positions 71, 72, 75, and 76 must be blank.

For the use of EXFMT with multiple device files, see the descriptions of the READ (by format name) and WRITE operations.

For more information, see "File Operations" on page 453.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
F*Flename++IPEASFRLen+LKlen+AIDevice+.Keywords++++++++++++++++++++++++++
 *
 * PROMTD is a WORKSTN file which prompts the user for an option.
 * Based on what user enters, this program executes different
 * subroutines to add, delete, or change a record.
 *
FPROMTD    CF   E             WORKSTN

 /free
    // If user enters F3 function key, indicator *IN03 is set
    // on and the do while loop is exited.
    dow not *in03;

       // EXFMT writes out the prompt to the screen and expects user to
       // enter an option. SCR1 is a record format name defined in the
       // WORKSTN file and OPT is a field defined in the record.
       exfmt SCR1;
       select;
       when opt = 'A';
          exsr AddRec;
       when opt = 'D';
          exsr DelRec;
       when opt = 'C';
          exsr ChgRec;
       endsl;
    enddo;
    do_something ();
    do_more_stuff ();
 /end-free
```

Figure 320. EXFMT Operation

```
  * DDS for display file MYDSPF
A          R REC
A            QUESTION      40A  O  5  2
A            NAME          20A  I  7  5
A            CITY          20A  B  8  5
 * RPG program using MYDSPF
Fmydspf    cf   e             workstn
 * Define a data structure for use with EXFMT REC
D recDs           ds                  likerec(rec : *all)
 /free
        // Set the output-capable fields
        recDs.question = 'What is your name?';
        recDs.city = 'Toronto';
        // Show the screen to the user
        exfmt rec recDs;
        // Use the input-capable fields
        // Since the "city" field is both input and output
        // capable, its value may have changed during EXFMT
        dsply ('Hello ' + recDs.name + ' in ' + recDs.city);
```

Figure 321. Using a result data structure with EXFMT

EXSR (Invoke Subroutine)

Free-Form Syntax	EXSR *subroutine-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
EXSR		subroutine-name				

The EXSR operation causes the RPG IV subroutine named in the *subroutine-name* operand to be processed. The 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. Whenever it appears, the subroutine that is named is processed. After operations in the subroutine are processed, the statement following the EXSR operation is processed, except when a GOTO within the subroutine is given to a label outside the subroutine or when the subroutine is an exception/error subroutine specified by the *return-point* operand of the ENDSR operation.

*PSSR used in the *subroutine-name* operand specifies that the program exception/error subroutine is to be processed. *INZSR used in the *subroutine-name* operand specifies that the program initialization subroutine is to be processed.

See “Coding Subroutines” on page 472, “Subroutine Operations” on page 472, or “Compare Operations” on page 445 for more information.

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 code will return one of:

- 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

to the field specified in the result field.

The Date, Time or Timestamp from which the information is required, is specified in factor 2, followed by the duration code. The entry specified in factor 2 can be a field, subfield, table element, or array element. The duration code must be consistent with the Data type of factor 2. See "Date Operations" on page 449 for valid duration codes.

Factor 1 must be blank.

The result field can be any numeric or character field, subfield, array/table element. The result field is cleared before the extracted data is assigned. For a character result field, the data is put left adjusted into the result field.

Note: 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. See Figure 99 on page 215 for an example of obtaining the Julian format.

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/Errors" on page 96.

For more information, see "Date Operations" on page 449.

```
     D LOGONDATE       S               D
     D DATE_STR        S             15
     D MONTHS          S              8    DIM(12) CTDATA
     C*0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
      * Move the job date to LOGONDATE.  By default, LOGONDATE has an *ISO
      * date format, which contains a 4-digit year.  *DATE also contains a
      * 4-digit year, but in a different format, *USA.
     C     *USA          MOVE      *DATE         LOGONDATE
      *
      * Extract the month from a date field to a 2-digit field
      * that is used as an index into a character array containing
      * the names of the months.  Then extract the day from the
      * timestamp to a 2-byte character field which can be used in
      * an EVAL concatenation expression to form a string.
      * For example, if LOGONDATE is March 17, 1996, LOGMONTH will
      * contain 03, LOGDAY will contain 17, and DATE_STR will contain
      * 'March 17'.
     C                   EXTRCT    LOGONDATE:*M  LOGMONTH          2 0
     C                   EXTRCT    LOGONDATE:*D  LOGDAY            2
     C                   EVAL      DATE_STR = %TRIMR(MONTHS(LOGMONTH))
     C                                        + ' ' + LOGDAY
     C                   SETON                                        LR
** CTDATA MONTHS
January
February
March
April
May
June
July
August
September
October
November
December
```

Figure 322. EXTRCT Operation

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 a primary, secondary, or full procedural file. The FEOD function differs, depending on the file type and device. (For an explanation of how FEOD differs per file type and device, see the iSeries Information Center database and file systems category).

FEOD differs from the CLOSE operation: the program is not disconnected from the device or file; the file can be used again for subsequent file operations without an explicit OPEN operation being specified to the file.

You can specify conditioning indicators. 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 or SEQ file that uses blocking (see “Blocking Considerations” on page 91). 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 79.

To process any further sequential operations to the file after the FEOD operation (for example, READ or READP), you must reposition the file.

For more information, see “File Operations” on page 453.

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

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 469.

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 708 and "ITER (Iterate)" on page 703 for information on how those operations affect a FOR operation.

For more information, see "Structured Programming Operations" on page 469.

```
*..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 323. Examples of the FOR Operation

FORCE (Force a Certain File to Be Read Next Cycle)

Free-Form Syntax	FORCE *file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
FORCE		file-name				

The FORCE operation allows selection of the file from which the next record is to be read. It can be used only for primary or secondary files.

The *file-name* operand must be the name of a file from which the next record is to be selected.

If the FORCE operation is processed, the record is read at the start of the next program cycle. If more than one FORCE operation is processed during the same program cycle, all but the last is ignored. FORCE must be issued at *detail* time, not total time.

FORCE operations override the multi-file processing method by which the program normally selects records. However, the first record to be processed is always selected by the normal method. The remaining records can be selected by FORCE operations. For information on how the FORCE operation affects match-field processing, see Figure 8 on page 33.

If FORCE is specified for a file that is at end of file, no record is retrieved from the file. The program cycle determines the next record to be read.

For more information, see “File Operations” on page 453.

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) another calculation operation in the program. A "TAG (Tag)" on page 828 operation names the destination of a GOTO operation. The TAG can either precede or follow the GOTO. Use a GOTO operation to specify a branch:

- From a detail calculation line to another detail calculation line
- From a total calculation line to another total calculation line
- From a detail calculation line to a total calculation line
- From a subroutine to a TAG or ENDSR within the same subroutine
- From a subroutine to a detail calculation line or to a total calculation line.

A GOTO within a subroutine in the cycle-main procedure can be issued to a TAG within the same subroutine, detail calculations or total calculations. 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.

Branching from one part of the RPG IV logic cycle to another may result in an endless loop. You are responsible for ensuring that the logic of your program does not produce undesirable results.

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 more information, see "Branching Operations" on page 439.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  If indicator 10, 15, or 20 is on, the program branches to
 *  the TAG label specified in the GOTO operations.
 *  A branch within detail calculations.
C   10              GOTO      RTN1
 *
 *  A branch from detail to total calculations.
C   15              GOTO      RTN2
 *
C     RTN1          TAG
 *
C                      :
C                      :
C:
C   20              GOTO      END
 *
C                      :
C                      :
C                      :
C     END           TAG
 *  A branch within total calculations.
CL1                 GOTO      RTN2
CL1                 :
CL1   RTN2          TAG
```

Figure 324. 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 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 IF operation are performed when the expression in the *indicator-expression* operand is true. For information on how operation extenders M and R are used, see “Precision Rules for Numeric Operations” on page 486.

For more information, see “Structured Programming Operations” on page 469.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++..
C                              Extended-factor2-continuation+++++++++++++++
 * The operations controlled by the IF operation are performed
 * when the expression is true.  That is A is greater than 10 and
 * indicator 20 is on.
C
C                   IF        A>10 AND *IN(20)
C                   :
C                   ENDIF
 *
 * The operations controlled by the IF operation are performed
 * when Date1 represents a later date then Date2
C
C                   IF        Date1 > Date2
C                   :
C                   ENDIF
 *
```

Figure 325. IF Operation

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)" on page 613 and "ORxx (Or)" on page 761 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 469.)

You can use conditioning indicators. 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 (Else)" on page 671 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 section on compiler listings in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for an explanation of how to indent statements in the source listing.

For more information, see "Compare Operations" on page 445 or "Structured Programming Operations" on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  If FLDA equals FLDB, the calculation after the IFEQ operation
 *  is processed.  If FLDA does not equal FLDB, the program
 *  branches to the operation immediately following the ENDIF.
C
C     FLDA          IFEQ      FLDB
C                       :
C                       :
C                       :
C                   ENDIF
C
 *  If FLDA equals FLDB, the calculation after the IFEQ operation
 *  is processed and control passes to the operation immediately
 *  following the ENDIF statement.  If FLDA does not equal FLDB,
 *  control passes to the ELSE statement and the calculation
 *  immediately following is processed.
C
C     FLDA          IFEQ      FLDB
C                       :
C                       :
C                       :
C                   ELSE
C                       :
C                       :
C                       :
C                   ENDIF
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  If FLDA is equal to FLDB and greater than FLDC, or if FLDD
 *  is equal to FLDE and greater than FLDF, the calculation
 *  after the ANDGT operation is processed.  If neither of the
 *  specified conditions exists, the program branches to the
 *  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                       :
C                   ENDIF
```

Figure 326. 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 and optionally allows you to specify whether the data area is to be locked from update by another program. For a data area to be retrieved by the IN operation, it must be specified in the result field of an *DTAARA DEFINE statement or using the DTAARA keyword on the Definition specification. (See "DEFINE (Field Definition)" on page 651 for information on *DTAARA DEFINE operation and the Definition Specification for information on the DTAARA keyword).

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.

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 RPG IV program implicitly unlocks the data area when the program ends

*LOCK cannot be specified when the *data-area-name* operand is the name of the local data area or the Program Initialization Parameters (PIP) data area.

You can specify a *LOCK IN statement for a data area that the program has locked. When *data-area-name* is not specified, the lock status is the same as it was before the data area was retrieved: If it was locked, it remains locked; if unlocked, it remains unlocked.

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 an error occurs on the retrieval of a data area (for example, a data area can be retrieved but cannot be locked), an error occurs on the IN operation and the RPG IV exception/error handling routine receives control. If a message is issued to the requester, the message identifies the data area in error.

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/Errors" on page 96.

On a fixed-form calculation, positions 71-72 and 75-76 must be blank.

For further rules for the IN operation, see "Data-Area Operations" on page 448.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 * Define Data areas
D TotAmt          s              8p 2 dtaara
D TotGrs          s             10p 2 dtaara
D TotNet          s             10p 2 dtaara

 *  TOTAMT, TOTGRS, and TOTNET are defined as data areas.  The IN
 *  operation retrieves all the data areas defined in the program
 *  and locks them.  The program processes calculations, and at
 *  LR time it writes and unlocks all the data areas.
 *  The data areas can then be used by other programs.

 /free

      in *lock *dtaara;
      TotAmt = TotAmt + Amount;
      TotGrs = TotGrs + Gross;
      TotNet = TotNet + Net;

 /end-free
 * To start total calcs, code a fixed format calc statement with a
 * level entry specified.
CL0   total_calcs   tag
 /free

      if *inlr
         out *dtaara
      endif
 /end-free
```

Figure 327. 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 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 present 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 (Leave a Do/For Group)" on page 708 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 439 or "Structured Programming Operations" on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following example uses a DOU loop containing a DOW loop.
 * The IF statement checks indicator 01.  If indicator 01 is ON,
 * the LEAVE operation is executed, transferring control out of
 * the innermost DOW loop to the Z-ADD instruction.  If indicator
 * 01 is not ON, subroutine PROC1 is processed.  Then indicator
 * 12 is checked.  If it is OFF, ITER transfers control to the
 * innermost ENDDO and the condition on the DOW is evaluated
 * 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                   :
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following example uses a DOU loop containing a DOW loop.
 * The IF statement checks indicator 1. If indicator 1 is ON, the
 * MOVE operation is executed, followed by the LEAVE operation,
 * transferring control from the innermost DOW loop to the Z-ADD
 * instruction.  If indicator 1 is not ON, ITER transfers control
 * to the innermost ENDDO and the condition on the DOW is
 * 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 328. ITER Operation

KFLD (Define Parts of a Key)

Free-Form Syntax	(not allowed - use %KDS)

Code	Factor 1	Factor 2	Result Field	Indicators		
KFLD		Indicator	Key field			

The KFLD operation is a declarative operation that indicates that a field is part of a search argument identified by a KLIST name.

The KFLD operation can be specified anywhere within calculations, including total calculations. The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. Conditioning indicator entries (positions 9 through 11) are not permitted.

KFLDs can be global or local. A KLIST in a cycle-main procedure can have only global KFLDs associated with it. A KLIST in a subprocedure can have local and global KFLDs. For more information, see “Scope of Definitions” on page 24.

Factor 2 can contain an indicator for a null-capable key field if ALWNULL(*USRCTL) is specified as a keyword on a control specification or as a command parameter.

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 223 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 329 on page 707 shows an example of the KLIST operation with KFLD operations.

Figure 105 on page 225 illustrates how keyed operations are used to position and retrieve records with null keys.

For more information, see “Declarative Operations” on page 452.

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 is a declarative operation that gives a name to a list of KFLDs. This list can be used as a search argument to retrieve records from files that have a composite key.

You can specify a KLIST anywhere within calculations. The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. Conditioning indicator entries (positions 9 through 11) are not permitted. Factor 1 must contain a unique name.

Remember the following when specifying a KLIST operation:

- If a search argument is composed of more than one field (a composite key), you must specify a KLIST with multiple KFLDs.
- A KLIST name can be specified as a search argument only for externally described files.
- A KLIST and its associated KFLD fields can appear anywhere in calculations.
- A KLIST must be followed immediately by at least one KFLD.
- A KLIST is ended when a non-KFLD operation is encountered.
- A KLIST name can appear in factor 1 of a CHAIN, DELETE, READE, READPE, SETGT, or SETLL operation.
- 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.
- A KLIST in a cycle-main procedure can have only global KFLDs associated with it. A KLIST in a subprocedure can have local and global KFLDs. For more information, see "Scope of Definitions" on page 24.

For more information, see "Declarative Operations" on page 452.

```
*...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....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The KLIST operation indicates the name, FILEKY, by which the
 *  search argument can be specified.
 *
C     FILEKY        KLIST
C                   KFLD                    DEPT
C                   KFLD                    SHIFT
C                   KFLD                    CLOCK#
```

The following diagram shows what the search argument looks like. The fields DEPT, SHIFT, and CLOCK# are key fields in this record.

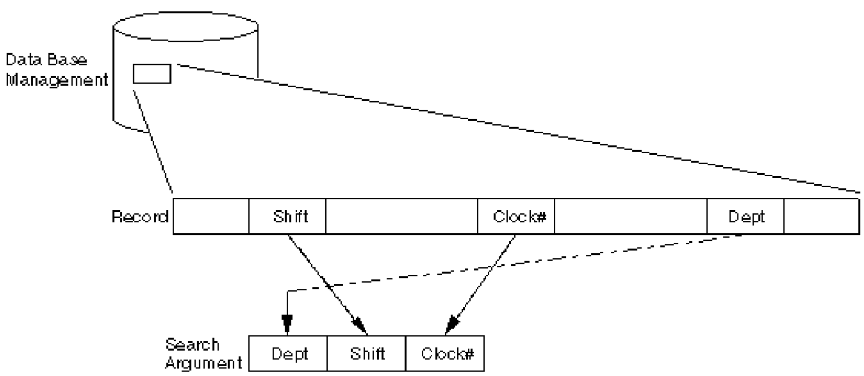


Figure 329. KLIST and KFLD Operations

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 (Iterate)" on page 703 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 439 or "Structured Programming Operations" on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following example uses an infinite loop.  When the user
 * types 'q', control transfers to the LEAVE operation, which in
 * turn transfers control out of the loop to the Z-ADD operation.
 *
C     2             DOWNE     1
C                   :
C                   IF        ANSWER = 'q'
C                   LEAVE
C                   ENDIF
C                   :
C                   ENDDO
C                   Z-ADD     A             B
 *
 * The following example uses a DOUxx loop containing a DOWxx.
 * The IF statement checks indicator 1. If it is ON, indicator
 * 99 is turned ON, control passes to the LEAVE operation and
 * out of the inner DOWxx loop.
 *
 * A second LEAVE instruction is then executed because indicator 99
 * is ON, which in turn transfers control out of the DOUxx loop.
 *
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 330. 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 472.

```
CL0N01Factor1+++++++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 331. 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 causes a search to be made for a particular element in an array or table. Factor 1 is the search argument (data for which you want to find a match in the array or table named). It can 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:

Character data
If ALTSEQ(*EXT) is specified on the control specification, the alternate collating sequence is used for character LOOKUP, unless either factor 1 or factor 2 was defined with ALTSEQ(*NONE) on the definition specification. If ALTSEQ(*SRC) or no alternate sequence is specified, character LOOKUP does not use the alternate sequence.

Graphic and UCS-2 data
The comparison is hexadecimal; the alternate collating sequence is not used in any circumstance.

Numeric data
The decimal point is ignored in numeric data, except when the array or table in Factor 2 is of type float.

Other data types
The considerations for comparison described in “Compare Operations” on page 445 apply to other types.

If a table is named in factor 1, the search argument used is the element of the table last selected in a LOOKUP operation, or it is the first element of the table if a previous LOOKUP has not been processed. The array or table to be searched is specified in factor 2.

For a table LOOKUP, the result field can contain the name of a second table from which an element (corresponding positionally with that of the first table) can be retrieved. The name of the second table can be used to reference the element retrieved. The result field must be blank if factor 2 contains an array name.

Resulting indicators specify the search condition for LOOKUP. One must be specified in positions 71 through 76 first to determine the search to be done and then to reflect the result of the search. 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. If such a higher entry is found, the high indicator is set on. For example, if an ascending array contains the values A B C C C D E, and the search argument is B, then the first C will satisfy the search. If a descending array contains E D C C C B A, and the search argument is B, then the last C will satisfy the search. If an entry higher than the search argument is not found in the array or table, then the search is unsuccessful.
- *Low (73-74):* Instructs the program to find the entry that is nearest to, yet lower in sequence than, the search argument. If such a lower entry is found, the low indicator is set on. For example, if an ascending array contains the values A B C C C D E, and the search argument is D, then the last C will satisfy the search. If a descending array contains E D C C C B A, and the search argument is D, then the first C will satisfy the search. If an entry lower than the search argument is not found in the array or table, then the search is unsuccessful.
- *Equal (75-76):* Instructs the program to find the entry equal to the search argument. The first equal entry found sets the equal indicator on. If an entry equal to the search argument is not found, then the search is unsuccessful.

When you use the LOOKUP operation, remember:

- The search argument and array or table 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.
- When LOOKUP is processed on an array and an index is used, the LOOKUP begins with the element specified by the index. The index value is set to the position number of the element located. An error occurs if the index is equal to zero or is higher than the number of elements in the array when the search begins. The index is set equal to one if the search is unsuccessful. If the index is a named constant, the index value will not change.
- A search can be made for high, low, high and equal, or low and equal only if a sequence is specified for the array or table on the definition specifications with the ASCEND or DESCEND keywords.
- No resulting indicator is set on if the search is not successful.
- If only an equal indicator (positions 75-76) is used, the LOOKUP operation will search the entire array or table. If your array or table is in ascending sequence and you want only an equal comparison, you can avoid searching the entire array or table by specifying a high indicator.
- The LOOKUP operation can produce unexpected results when the array is not in ascending or descending sequence.
- A LOOKUP operation to a dynamically allocated array without all defined elements allocated may cause errors to occur.

For more information, see “Array Operations” on page 438.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  In this example, the programmer wants to know which element in
 *  ARY the LOOKUP operation locates.  The Z-ADD operation sets the
 *  field X to 1.  The LOOKUP starts at the element ARY that is
 *  indicated by field X and continues running until it finds the
 *  first element equal to SRCHWD.  The index value, X, is set to
 *  the position number of the element located.
C
C                   Z-ADD     1              X                 3 0
C     SRCHWD        LOOKUP    ARY(X)                                  26
C
 *  In this example, the programmer wants to know if an element
 *  is found that is equal to SRCHWD.  LOOKUP searches ARY until it
 *  finds the first element equal to SRCHWD.  When this occurs,
 *  indicator 26 is set on and %EQUAL is set to return '1'.
C
C     SRCHWD        LOOKUP    ARY                                     26
C
 *  The LOOKUP starts at a variable index number specified by
 *  field X.  Field X does not have to be set to 1 before the
 *  LOOKUP operation.  When LOOKUP locates the first element in
 *  ARY equal to SRCHWD, indicator 26 is set on and %EQUAL is set
 *  to return '1'.  The index value, X, is set to the position
 *  number of the element located.
 *
C
C     SRCHWD        LOOKUP    ARY(X)                                  26
```

Figure 332. 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 333. LOOKUP Operation with Subarrays

MHHZO (Move High to High Zone)

Free-Form Syntax	(not allowed - use the %BITAND and %BITOR built-in functions. See Figure 197 on page 504.)

Code	Factor 1	Factor 2	Result Field	Indicators		
MHHZO		Source field	Target field			

The MHHZO operation moves the zone portion of a character from the leftmost zone in factor 2 to the leftmost zone in the result field. Factor 2 and the result field must both be defined as character fields. For further information on the MHHZO operation, see “Move Zone Operations” on page 466.

The function of the MHHZO operation is shown in Figure 180 on page 467.

MHLZO (Move High to Low Zone)

Free-Form Syntax	(not allowed - use the %BITAND and %BITOR built-in functions. See Figure 197 on page 504.)

Code	Factor 1	Factor 2	Result Field	Indicators		
MHLZO		Source field	Target field			

The MHLZO operation moves the zone portion of a character from the leftmost zone in factor 2 to the rightmost zone in the result field. Factor 2 must be defined as a character field. The result field can be character or numeric data. For further information on the MHLZO operation, see “Move Zone Operations” on page 466.

The function of the MHLZO operation is shown in Figure 180 on page 467.

MLHZO (Move Low to High Zone)

Free-Form Syntax	(not allowed - use the %BITAND and %BITOR built-in functions. See Figure 197 on page 504.)

Code	Factor 1	Factor 2	Result Field	Indicators		
MLHZO		Source field	Target field			

The MLHZO operation moves the zone portion of a character from the rightmost zone in factor 2 to the leftmost zone in the result field. Factor 2 can be defined as a numeric field or as a character field, but the result field must be a character field. For further information on the MLHZO operation, see "Move Zone Operations" on page 466.

The function of the MLHZO operation is shown in Figure 180 on page 467.

MLLZO (Move Low to Low Zone)

Free-Form Syntax	(not allowed - use the %BITAND and %BITOR built-in functions. See Figure 197 on page 504.)

Code	Factor 1	Factor 2	Result Field	Indicators		
MLLZO		Source field	Target field			

The MLLZO operation moves the zone portion of a character from the rightmost zone in factor 2 to the rightmost zone in the result field. Factor 2 and the result field can be either character data or numeric data. For further information on the MLLZO, see “Move Zone Operations” on page 466.

The function of the MLLZO operation is shown in Figure 180 on page 467.

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.

Level indicators can be used on the MONITOR operation, to indicate that the MONITOR group is part of total calculations. For documentation purposes, you can also specify a level indicator on an ON-ERROR or ENDMON operation but this level indicator will be ignored.

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 452.

```
 * 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 334. 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 data, 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 or target of the operation. For information on the formats that can be used see "Date Data Type" on page 206, "Time Data Type" on page 208, and "Timestamp Data Type" on page 210.

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 data, 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. A MOVE operation does not change the length of a variable-length result field. For examples, see Figures 339 to 344. The graphic literals in this examples are not valid graphic literals. See "Graphic Format" on page 183 for more information.

The tables which appear following the examples, show how data is moved from factor 2 to the result field. For further information on the MOVE operation, see "Move Operations" on page 460 or "Conversion Operations" on page 447.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
 *  Control specification date format
H DATFMT(*ISO)
 *
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
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)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 * Move between Date fields. DATE_EUR will contain 24.03.1992
 *
C                   MOVE      DATE_YMD      DATE_EUR
 *
 * Convert numeric value in ddmmyy format into a *ISO Date.
 * DATE_ISO will contain 1991-09-21 after each of the 2 moves.
C     *DMY          MOVE      210991        DATE_ISO
C     *DMY          MOVE      NUM_DATE1     DATE_ISO
 *
 * Move a character value representing a *MDY date to a *JIS Date.
 * DATE_JIS will contain 1953-02-01 after each of the 2 moves.
C     *MDY/         MOVE      '02/01/53'    DATE_JIS
C     *MDY/         MOVE      CHAR_DATE     DATE_JIS
 *
 *  Move a date field to a character field, using the
 *  date format and separators based on the job attributes
C     *JOBRUN       MOVE (P)  DATE_JIS      CHAR_DATE
 *
 *  Move a date field to a numeric field, using the
 *  date format based on the job attributes
 *
 *  Note:  If the job format happens to be *JUL, the date will
 *         be placed in the rightmost 5 digits of NUM_DATE1.
 *         The MOVEL operation might be a better choice.
 *
C     *JOBRUN       MOVE (P)  DATE_JIS      NUM_DATE1
 *
 *  DATE_USA will contain 12-31-9999
C                   MOVE      *HIVAL        DATE_USA
 *
 *  Execution error, resulting in error code 114.  Year is not in
 *  1940-2039 date range.  DATE_YMD will be unchanged.
C                   MOVE      DATE_USA      DATE_YMD
 *
 *  Move a *EUR date field to a numeric field that will
 *  represent a *CMDY date.  NUM_DATE2 will contain 2082697
 *  after the move.
C     *CMDY         MOVE      DATE_EUR      NUM_DATE2
 *
 *  Move a character value representing a *LONGJUL date to
 *  a *YMD date.  DATE_YMD will be 39/06/15 after the move.
C     *LONGJUL      MOVE      CHAR_LONGJUL  DATE_YMD
```

Figure 335. MOVE Operation with Date

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
 *  Specify default format for date fields
H DATFMT(*ISO)
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
D date_USA      S               D   DATFMT(*USA)
D datefld       S               D
D timefld       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
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 * Move a character value representing a *MDY date to a D(Date) value.
 * *MDY0 indicates that the character date in Factor 2 does not
 * contain separators.
 * datefld will contain 1996-04-15 after the move.
C     *MDY0         MOVE      chr_dateA     datefld
 * Move a field containing a T(Time) value to a character value in the
 * *EUR format.  *EUR0 indicates that the result field should not
 * contain separators.
 * chr_time will contain '142310' after the move.
C     *EUR0         MOVE      timefld       chr_time
 * Move a character value representing a *CYMD date to a *USA
 * Date. Date_USA will contain 08/07/1961 after the move.
 * 0 in *CYMD indicates that the character value does not
 * contain separators.
C     *CYMD0        MOVE      chr_dateB     date_USA
```

Figure 336. MOVE Operation with Date and Time without Separators

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
 *  Control specification DATEDIT format
 *
H DATEDIT(*MDY)
 *
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
 *
 * Set the timestamp Jobstart with the job start Date and Time
 *
 * Factor 1 of the MOVE *DATE (*USA = MMDDYYYY) is consistent
 * with the value specified for the DATEDIT keyword on the
 * control specification, since DATEDIT(*MDY) indicates that
 * *DATE is formatted as MMDDYYYY.
 *
 * Note:   It is not necessary to specify factor 1 with *DATE or
 *         UDATE.
 *
CL0N01Factor1+++++++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
 *
 * After the following C specifications are performed, the field
 * stampchar will contain '1991-10-24-05.17.23.000000'.
 *
 * First assign a timestamp the value of a given time+15 minutes and
 * given date + 30 days. Move tmstamp to a character field.
 * stampchar will contain '1991-10-24-05.17.23.000000'.
 *
C                   ADDDUR    15:*minutes   Timebegin
C                   ADDDUR    30:*days      Datebegin
C                   MOVE      Timebegin     TmStamp
C                   MOVE      Datebegin     TmStamp
C                   MOVE      TmStamp       stampchar        26
 * Move the timestamp to a character field without separators. After
 * the move, STAMPCHAR will contain '      19911024051723000000'.
C     *ISO0         MOVE(P)   TMSTAMP       STAMPCHAR0
```

Figure 337. MOVE Operation with Timestamp

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVE between graphic and character fields
 *
D char_fld1      S             10A   inz('oK1K2K3  i')
D dbcs_fld1      S              4G
D char_fld2      S             10A   inz(*ALL'Z')
D dbcs_fld2      S              3G   inz(G'oK1K2K3i')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
 * Value of dbcs_fld1 after MOVE operation is 'K1K2K3  '
 * Value of char_fld2 after MOVE oepration is 'ZZoK1K2K3i'
 *
C                   MOVE      char_fld1     dbcs_fld1
C                   MOVE      dbcs_fld2     char_fld2
```

Figure 338. MOVE between character and graphic fields

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVE from variable to variable length
 * for character fields
 *
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
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE      var15a        var5a
 * var5a = 'ABFGH' (length=5)
C                   MOVE      var10a        var5b
 * var5b = '56789' (length=5)
C                   MOVE      var5c         var15a
 * var15a = 'CDE' (length=3)
C                   MOVE      var10b        var15b
 * var15b = 'BNM' (length=3)
C                   MOVE      var15c        var10b
 * var10b = 'YUIOPAS' (length=7)
```

Figure 339. MOVE from a variable-length field to variable-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVE from variable to fixed length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE      var5          fix5a
 * fix5a = 'ABCDE'
C                   MOVE      var10         fix5b
 * fix5b = '56789'
C                   MOVE      var15         fix5c
 * fix5c = 'MNFGH'
```

Figure 340. MOVE from a variable-length field to a fixed-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVE from fixed to variable length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE      fix10         var5
 * var5 = 'UVWXY' (length=5)
C                   MOVE      fix5          var10
 * var10 = '01234.....' (length=10)
C                   MOVE      fix10         var15
 * var15 = 'STUVWXY' (length=7)
```

Figure 341. MOVE from a fixed-length field to a variable-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVE(P) from variable to variable length
 * for character fields
 *
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
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE(P)   var15a        var5a
 * var5a = '  FGH' (length=5)
C                   MOVE(P)   var10         var5b
 * var5b = '56789' (length=5)
C                   MOVE(P)   var5c         var15b
 * var15b = 'CDE' (length=3)
C                   MOVE(P)   var10         var15c
 * var15c = '789' (length=3)
```

Figure 342. 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++++++++++++++++++++
 *
 * Example of MOVE(P) from variable to fixed length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE(P)   var5          fix5a
 * fix5a = 'ABCDE'
C                   MOVE(P)   var10         fix5b
 * fix5b = '56789'
C                   MOVE(P)   var15         fix5c
 * fix5c = '  FGH'
```

Figure 343. 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++++++++++++++++++++
 *
 * Example of MOVE(P) from fixed to variable length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVE(P)   fix10         var5
 * var5 = 'UVWXY' (length=5 before and after)
C                   MOVE(P)   fix10         var10
 * var10 = 'PQRSTUVWXY' (length=10 before and after)
C                   MOVE(P)   fix10         var15a
 * var15a = '   PQRSTUVWXY' (length=13 before and after)
C                   MOVE(P)   fix10         var15b
 * var15b = 'UVWXY' (length=5 before and after)
```

Figure 344. MOVE(P) from a fixed-length field to a variable-length field

Table 76. Moving a Character Field to a Date-Time Field. Factor 1 specifies the format of the Factor 2 entry

Factor 1 Entry	Factor 2 (Character)	Result Field	
		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 77. Moving a Numeric Field to a Date-Time Field. Factor 1 specifies the format of the Factor 2 entry

Factor 1 Entry[1]	Factor 2 (Numeric)	Result Field	
		Value	DTZ Type
*MDY	111975	75/323	D(*JUL)
*JUL	92114	23/04/92	D(*DMY)
*YMD	140128	01/28/2014	D(*USA)
*USA[2]	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)
*LONGJUL	2021189	08/07/2021	D(*EUR)
*USA	*DATE (092195)[3]	1995-09-21	D(*JIS)
Blank	*DATE (092195)[3]	1995-09-21	D(*JIS)
*MDY	UDATE (092195)[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[4]	19910914131256123456	1991-09-14-13.12.56.123456	Z(*ISO)

Notes:

1. A separator of zero (0) is not allowed in factor 1 for movement between date, time or timestamp fields and numeric classes.
2. Time format *USA is not allowed for movement between time 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.

Table 78. Moving a Date-Time Field to a Character Field

Factor 1 Entry	Factor 2		Result Field (Character)
	Value	DTZ Type	
*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)	21,05,36
*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(*MDY&);	108,03,07

Table 78. Moving a Date-Time Field to a Character Field (continued)

Factor 1 Entry	Factor 2		Result Field (Character)
	Value	DTZ Type	
*CYMD0	21/07/08	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 79. Moving a Date-Time Field to a Numeric Field

Factor 1 Entry	Factor 2		Result Field (Numeric)
	Value	DTZ Type	
*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	23 12 56	T(*HMS&);	231256
*EUR	11:00 AM	T(*USA)	110000
*JIS	11.10.07	T(*EUR)	111007
*HMS,	14:16:18	T(*JIS)	141618
*ISO	2045-10-27-23.34.59.123456	Z(*ISO)	20451027233459123456

Table 80. Moving Date-Time Fields to Date-Time Fields. Assume that the initial value of the timestamp is `1985-12-03-14.23.34.123456`.

Factor 1	Factor 2		Result Field	
	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)

Table 80. Moving Date-Time Fields to Date-Time Fields (continued). Assume that the initial value of the timestamp is `1985-12-03-14.23.34.123456`.

Factor 1	Factor 2		Result Field	
	Value	**DTZ Type**	**Value**	**DTZ Type**
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)

Table 81. 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        20A   INZ('ABCDEFGHIJabcdefghij')
```

Operation Code	Factor 2		Value of Result Field after move operation
	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 '

Table 82. 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)
```

Operation Code	Factor 2		Value of Result Field after move operation
	Value	**DTZ Type**	
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 83. 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.

Operation Code	Factor 2	Result Field	
		DTZ Type	**Value**
MOVE	11:**12:13:14**	T(*EUR)	12.13.14
MOVEL	**11:12:13**:14	T(*EUR)	11.12.13

Table 84. 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.

Operation Code	Factor 2	Result Field	
		DTZ Type	Value
MOVE	123406**18230323123420123456**	Z(*ISO)	1823-03-23-12.34.20.123456
MOVEL	**123406182303231234**20123456	Z(*ISO)	1234-06-18-23.03.23.123420

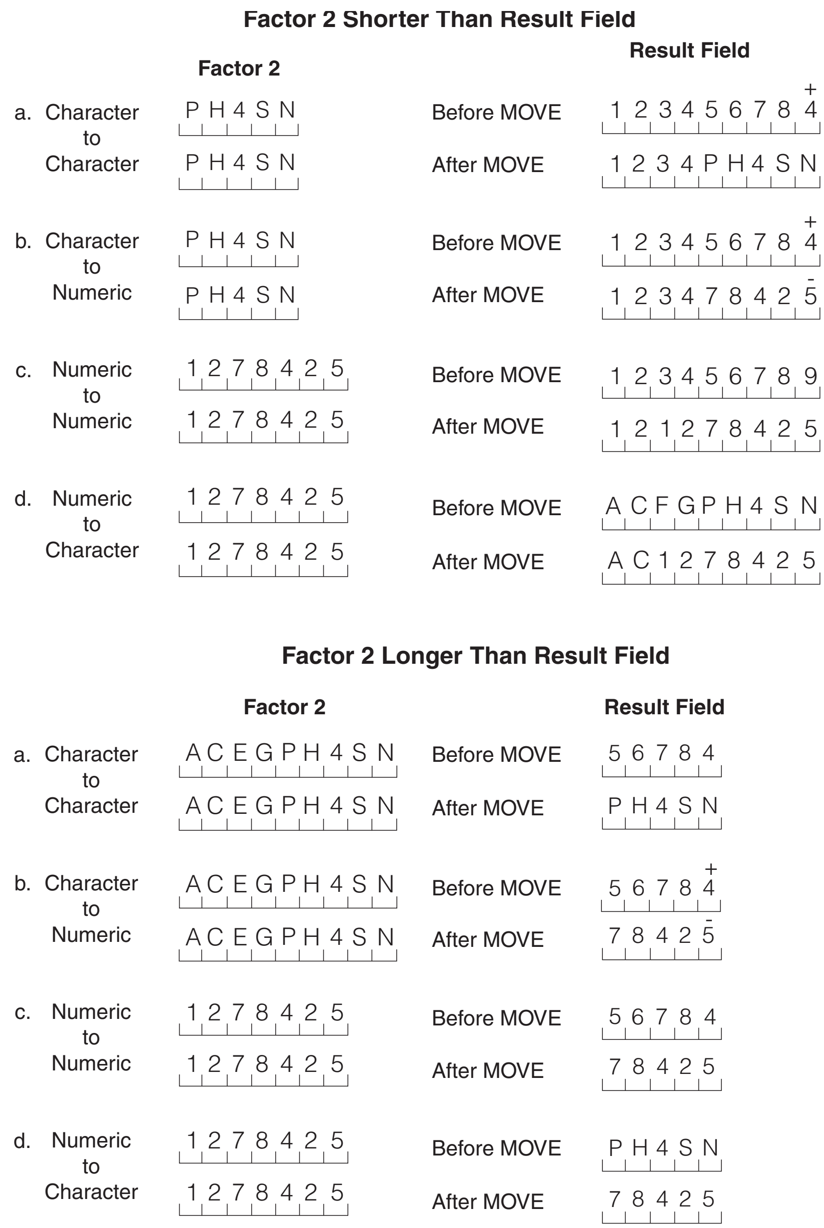


Figure 345. MOVE Operation (Part 1 of 2)

Factor 2 Shorter Than Result Field With P in Operation Extender Field

	Factor 2		Result Field
a. Character to Character	P H 4 S N	Before MOVE	1 2 3 4 5 6 7 8 $\overset{+}{4}$
	P H 4 S N	After MOVE	P H 4 S N
b. Character to Numeric	P H 4 S N	Before MOVE	1 2 3 4 5 6 7 8 $\overset{+}{4}$
	P H 4 S N	After MOVE	0 0 0 0 7 8 4 2 $\overset{-}{5}$
c. Numeric to Numeric	1 2 7 8 4 2 5	Before MOVE	1 2 3 4 5 6 7 8 9
	1 2 7 8 4 2 5	After MOVE	0 0 1 2 7 8 4 2 5
d. Numeric to Character	1 2 7 8 4 2 5	Before MOVE	A C F G P H 4 S N
	1 2 7 8 4 2 5	After MOVE	1 2 7 8 4 2 5

Factor 2 and Result Field Same Length

	Factor 2		Result Field
a. Character to Character	P H 4 S N	Before MOVE	5 6 7 8 4
	P H 4 S N	After MOVE	P H 4 S N
b. Character to Numeric	P H 4 S N	Before MOVE	5 6 7 8 4
	P H 4 S N	After MOVE	7 8 4 2 $\overset{-}{5}$
c. Numeric to Numeric	7 8 4 2 $\overset{-}{5}$	Before MOVE	A L T 5 F
	7 8 4 2 $\overset{-}{5}$	After MOVE	7 8 4 2 $\overset{-}{5}$
d. Numeric to Character	7 8 4 2 $\overset{-}{5}$	Before MOVE	A L T 5 F
	7 8 4 2 $\overset{-}{5}$	After MOVE	7 8 4 2 $\overset{-}{5}$

Note: $\overset{+}{4}$ = letter D , and $\overset{-}{5}$ = letter N.

Figure 345. MOVE Operation (Part 2 of 2)

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 the result field. (Certain restrictions apply when moving numeric values.) Factor 2 or the result field must contain an array. Factor 2 and the result field cannot specify the same array even if the array is indexed. You can:

- 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, all indicators affected by the MOVEA operation are noted in the cross-reference listing.

The coding for and results of MOVEA operations are shown in Figure 346 on page 735.

For more information, see “Array Operations” on page 438, “Move Operations” on page 460, or “Date Operations” on page 449.

Character, graphic, and UCS-2 MOVEA Operations

Both factor 2 and the result field must be the same type - either character, graphic, or UCS-2. Graphic or UCS-2 CCSIDs must be the same, unless one of the CCSIDs is 65535, or in the case of graphic fields, CCSID(*GRAPH: *IGNORE) was specified on the control specification.

On a character, graphic, or UCS-2 MOVEA operation, 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; therefore, the MOVEA operation could end in the middle of an array element. Variable-length arrays are not allowed.

Numeric MOVEA Operations

Moves are only valid between fields and array elements with the same numeric length defined. Factor 2 and the result field entries can specify 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 but need not be the same between factor 2 and the result field.

Factor 2 can contain a numeric literal if the result field entry specifies 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 or array element specified in the result field.

Decimal positions are ignored during the move and need not correspond. Numeric values are not converted to account for the differences in the defined number of decimal places.

The figurative constants *BLANK, *ALL, *ON and *OFF are not valid in factor 2 of a MOVEA operation on a numeric array.

General MOVEA Operations

If you need to use a MOVEA operation in your application, but restrictions on numeric MOVEA operations prevent you, you might be able to use character MOVEA operations. If the numeric array is in zoned decimal format:

- Define the numeric array as a subfield of a data structure
- Redefine the numeric array in the data structure as a character array.

If a figurative constant is specified with MOVEA, the length of the constant generated is equal to the portion of the array specified. For figurative constants in numeric arrays, the element boundaries are ignored except for the sign that is put in each array element. Examples are:

- 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. Beginning with element X, the remainder of the array will contain 'XYZXYZXYZXYZ. . .'.

For character, graphic, UCS-2, and numeric MOVEA operations, you can specify a P operation extender to pad the result from the right.

For further information on the MOVEA operation, see "Move Operations" on page 460.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY
 *  Array-to-array move.  No indexing; different length array,
 *  same element length.
```

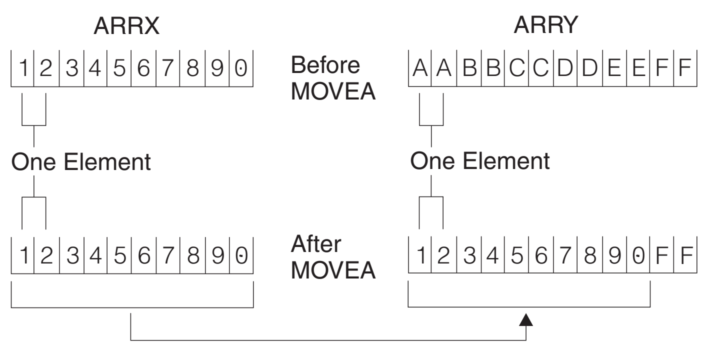


Figure 346. MOVEA Operation (Part 1 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY(3)
 *  Array-to-array move with index result field.
```

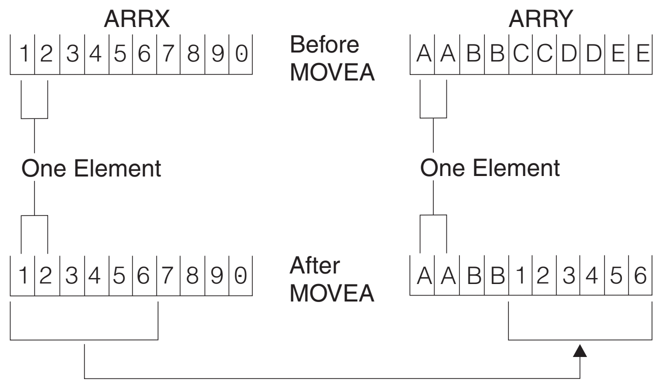


Figure 346. MOVEA Operation (Part 2 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX          ARRY
 *  Array-to-array move, no indexing and different length array
 *  elements.
```

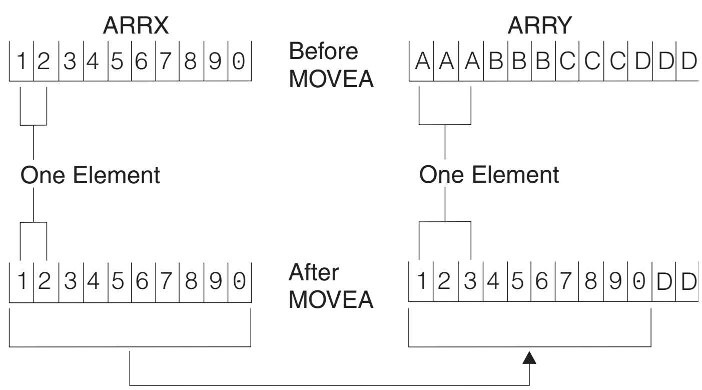


Figure 346. MOVEA Operation (Part 3 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRX(4)       ARRY
 *  Array-to-array move, index factor 2 with different length array
 *  elements.
```

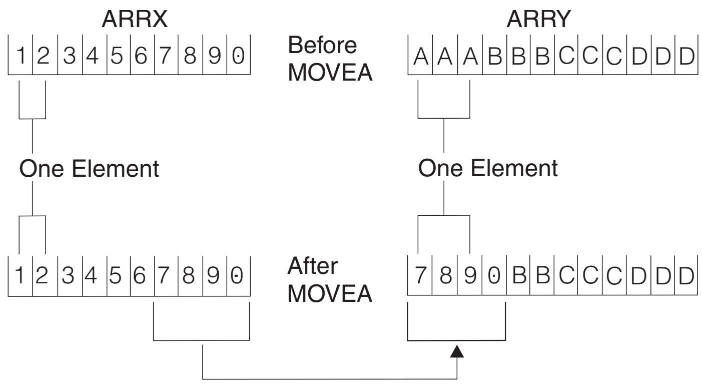


Figure 346. MOVEA Operation (Part 4 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     FIELDA        ARRY
 *  Field-to-array move, no indexing on array.
```

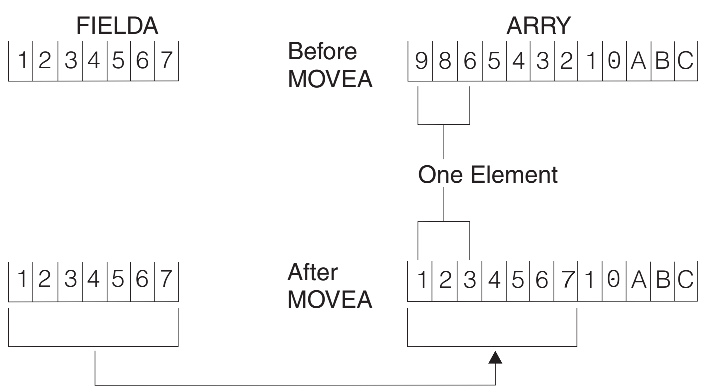


Figure 346. MOVEA Operation (Part 5 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * In the following example, N=3.  Array-to-field move with variable
 * indexing.
C                   MOVEA     ARRX(N)       FIELD
 *
```

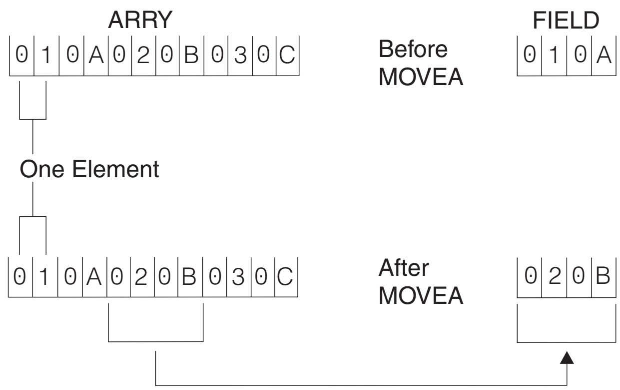


Figure 346. MOVEA Operation (Part 6 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA     ARRB          ARRZ
 *
 * An array-to-array move showing numeric elements.
```

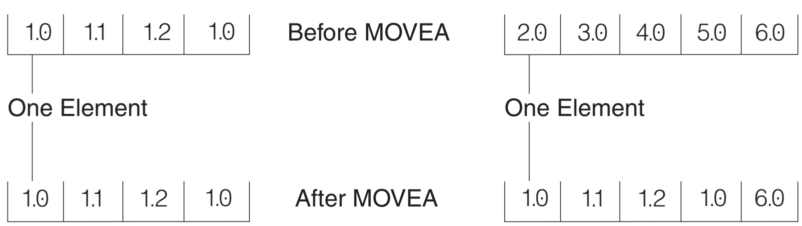


Figure 346. MOVEA Operation (Part 7 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRX          ARRY
 *  Array-to-array move with padding.  No indexing; different length
 *  array with same element length.
```

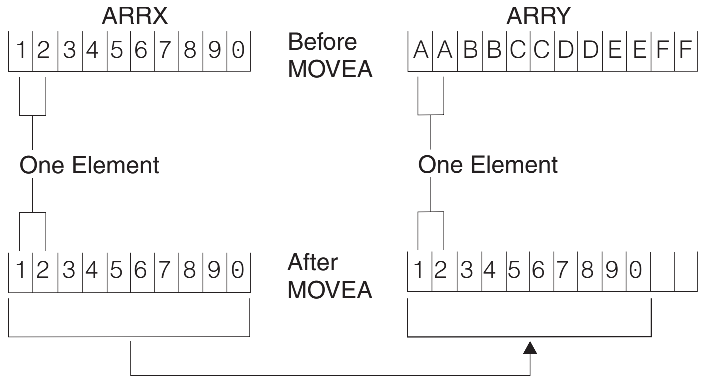


Figure 346. MOVEA Operation (Part 8 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRB          ARRZ
 *
 * An array-to-array move showing numeric elements with padding.
```

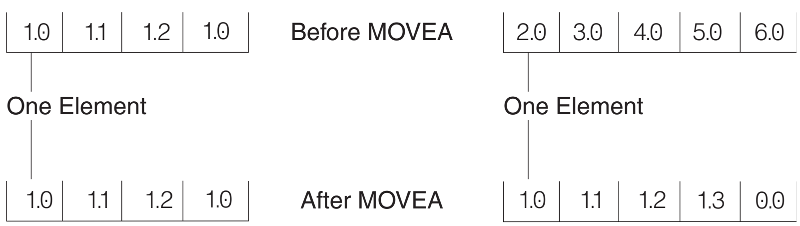


Figure 346. MOVEA Operation (Part 9 of 10)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVEA(P)  ARRX(3)       ARRY
 *  Array-to-array move with padding.  No indexing; different length
 *  array with different element length.
```

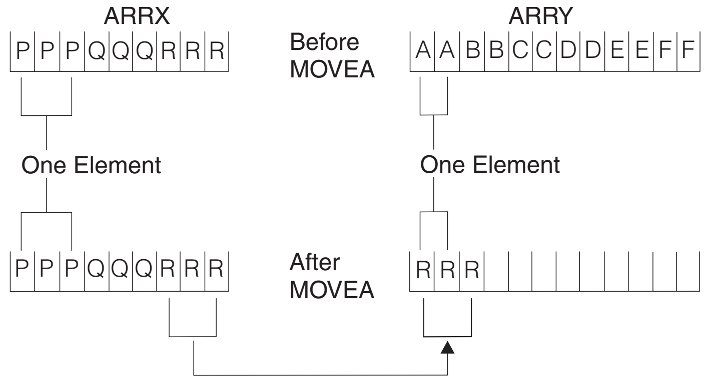


Figure 346. MOVEA Operation (Part 10 of 10)

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. 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 non-array field.

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.

Factor 1 can contain a date or time format to specify the format of a character or numeric field that is the source or target of the operation.For information on the formats that can be used see “Date Data Type” on page 206, “Time Data Type” on page 208, and “Timestamp Data Type” on page 210.

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 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 excess rightmost characters in the result field are unchanged, unless padding is specified.

Float numeric fields and literals are not allowed as Factor 2 or Result-Field entries.

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 since the character data may or may not contain shift characters and 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 with shift characters separating the modes

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, and contains matched SO/SI pairs.

The MOVEL operation is summarized in Figure 347 on page 744.

A summary of the rules for MOVEL operation for four conditions based on field lengths:

1. Factor 2 is the same length as the result field:
   a. If factor 2 and the result field are numeric, the sign is moved into the rightmost position.
   b. If factor 2 is numeric and the result field is character, the sign is moved into the rightmost position.
   c. 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 hexadecimal D (minus zone). However, if the zone from the rightmost position of factor 2 is not a hexadecimal D, 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.
   d. If factor 2 and the result field are character, all characters are moved.
   e. If factor 2 and the result field are both graphic or UCS-2, all graphic or UCS-2 characters are moved.
   f. If factor 2 is graphic and the result field is character, one graphic character will be lost, because 2 positions (bytes) in the character result field will be used to hold the SO/SI inserted by the compiler.
   g. If factor 2 is character and the result field is graphic, the factor 2 character data must be completely enclosed by one single pair of SO/SI. The SO/SI will be removed by the compiler before moving the data to the graphic result field.
2. Factor 2 is longer than the result field:
   a. If factor 2 and the result field are numeric, the sign from the rightmost position of factor 2 is moved into the rightmost position of the result field.
   b. If factor 2 is numeric and the result field is character, the result field contains only numeric characters.
   c. 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 hexadecimal D (minus zone). However, if the zone from the rightmost position of factor 2 is not a hexadecimal D, a positive zone is moved into the rightmost position of the result field. Other result field positions contain only numeric characters.
   d. If factor 2 and the result field are character, only the number of characters needed to fill the result field are moved.
   e. If factor 2 and the result field are graphic or UCS-2, only the number of graphic or UCS-2 characters needed to fill the result field are moved.
   f. If factor 2 is graphic and the result field is character, the graphic data will be truncated and SO/SI will be inserted by the compiler.
   g. If factor 2 is character and the result is graphic, the character data will be truncated. The character data must be completely enclosed by one single pair of SO/SI.
3. Factor 2 is shorter than the result field:
   a. 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.

   b. 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.
   c. If factor 2 is graphic and the result field is character, the SO/SI are added immediately before and after the graphic data. This may cause unbalanced SO/SI in the character field due to residual data in the field, but this is users' responsibility.
   d. Notice that when moving from a character to graphic field, the entire character field should be enclosed in SO/SI. For example, if the character field length is 8, the character data in the field should be "oAABBƀƀi" and not "oAABBiƀƀ".
4. Factor 2 is shorter than the result field and P is specified in the operation extender field:
   a. The move is performed as described above.
   b. The result field is padded from the right. See “Move Operations” on page 460 for more information on the rules for padding.

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. A MOVEL operation does not change the length of a variable-length result field. For examples, see Figures 350 to 355.

For further information on the MOVEL operation, see “Move Operations” on page 460, “Date Operations” on page 449, or “Conversion Operations” on page 447.

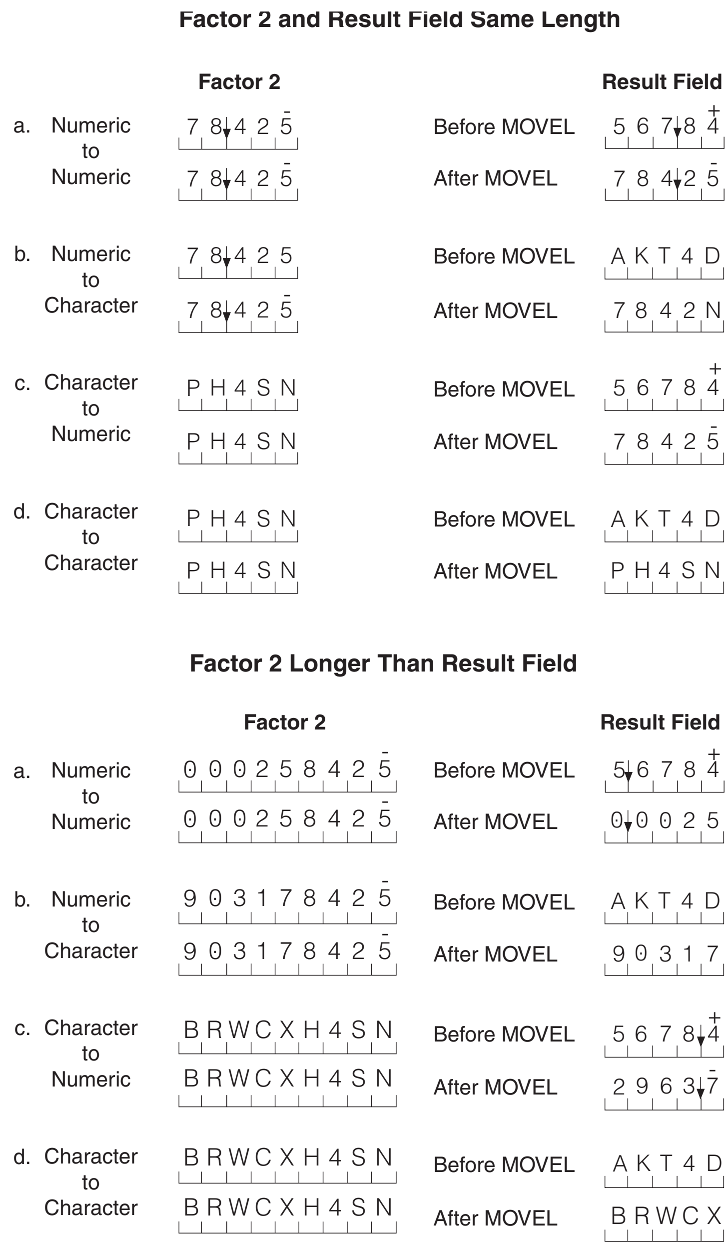


Figure 347. MOVEL Operation (Part 1 of 2)

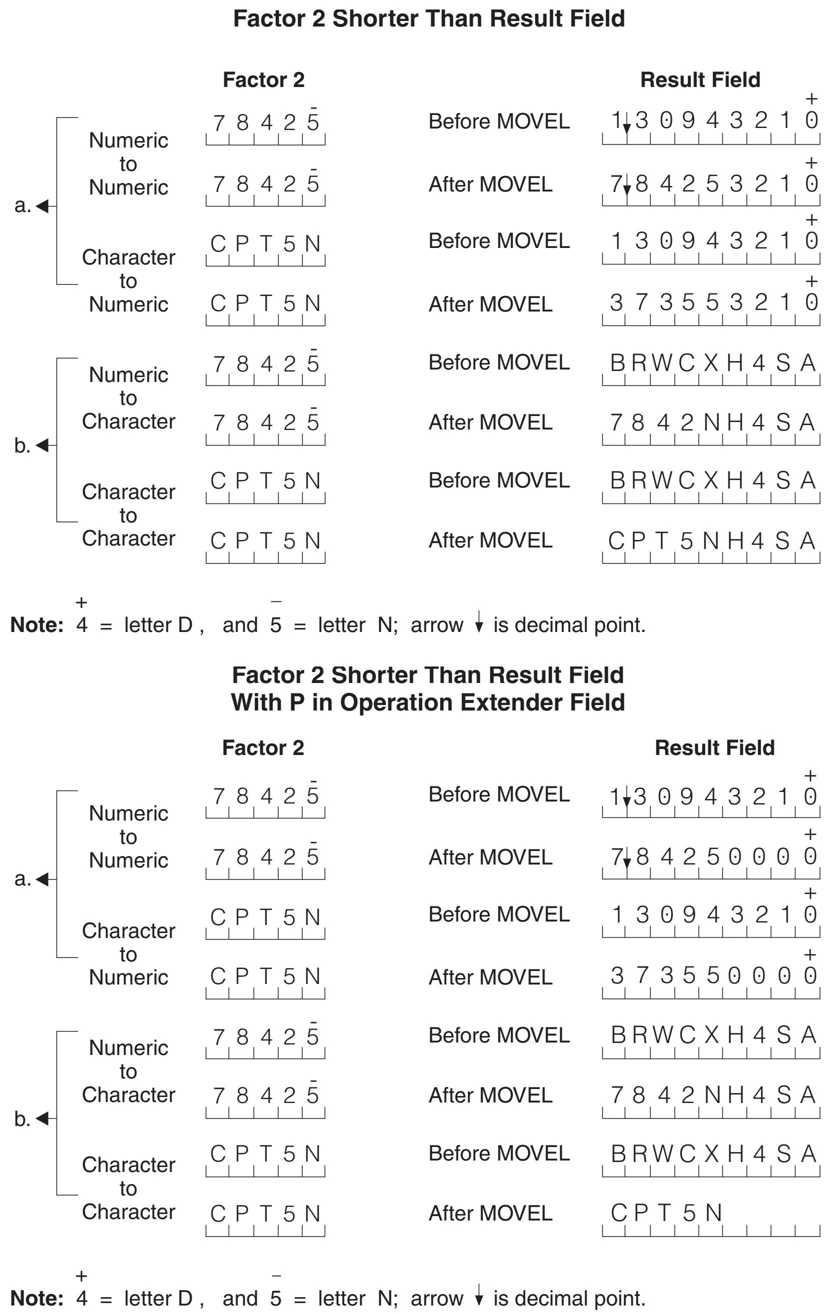


Figure 347. MOVEL Operation (Part 2 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
D
 *
 * Example of MOVEL between graphic and character fields
 *
D char_fld1      S              8A   inz(' ')
D dbcs_fld1      S              4G   inz('oAABBCCDDi')
D char_fld2      S              4A   inz(' ')
D dbcs_fld2      S              3G   inz(G'oAABBCCi')
D char_fld3      S             10A   inz(*ALL'X')
D dbcs_fld3      S              3G   inz(G'oAABBCCi')
D char_fld4      S             10A   inz('oAABBCC  i')
D dbcs_fld4      S              2G
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 *  The result field length is equal to the factor 2 length in bytes.
 *  One DBCS character is lost due to insertion of SO/SI.
 *  Value of char_fld1 after MOVEL operation is 'oAABBCCi'
 *
C                   MOVEL     dbcs_fld1     char_fld1
 *
 *  Result field length shorter than factor 2 length. Truncation occurs.
 *  Value of char_fld2 after MOVEL operation is 'oAAi'
 *
C                   MOVEL     dbcs_fld2     char_fld2
 *
 *  Result field length longer than factor 2 length. Example shows
 *  SO/SI are added immediately before and after graphic data.
 *  Before the MOVEL, Result Field contains 'XXXXXXXXXX'
 *  Value of char_fld3 after MOVEL operation is 'oAABBCCiXX'
 *
C                   MOVEL     dbcs_fld3     char_fld3
 *
 *  Character to Graphic MOVEL
 *  Result Field shorter than Factor 2. Truncation occurs.
 *  Value of dbcs_fld4 after MOVEL operation is 'AABB'
 *
C                   MOVEL     char_fld4     dbcs_fld4
```

Figure 348. MOVEL between character and graphic fields

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *
 * Example of MOVEL between character and date fields
 *
 *  Control specification date format
H DATFMT(*MDY)
 *
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++
D datefld         S              D   INZ(D'04/15/96')
D char_fld1       S             8A
D char_fld2       S            10A   INZ('XXXXXXXXXX')
D char_fld3       S            10A   INZ('04/15/96XX')
D date_fld3       S              D
D char_fld4       S            10A   INZ('XXXXXXXXXX')
D char_fld5       S             9A   INZ('015/04/50')
D date_fld2       S              D   INZ(D'11/16/10')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+H1LoEq..
 * Date to Character MOVEL
 * The result field length is equal to the factor 2 length.  Value of
 * char_fld1 after the MOVEL operation is '04/15/96'.
C     *MDY          MOVEL     datefld       char_fld1
 * Date to Character MOVEL
 * The result field length is longer than the factor 2 length.
 * Before MOVEL, result field contains 'XXXXXXXXXX'
 * Value of char_fld2 after the MOVEL operation is '04/15/96XX'.
C     *MDY          MOVEL     datefld       char_fld2
 * Character to Date MOVEL
 * The result field length is shorter than the factor 2 length.
 * Value of date_fld3 after the MOVEL operation is '04/15/96'.
C     *MDY          MOVEL     char_fld3     date_fld3
 * Date to Character MOVEL (no separators)
 * The result field length is longer than the factor 2 length.
 * Before MOVEL, result field contains 'XXXXXXXXXX'
 * Value of char_fld4 after the MOVEL operation is '041596XXXX'.
C     *MDY0         MOVEL     datefld       char_fld4
 *  Character to date MOVEL
 *  The result field length is equal to the factor 2 length.
 *  The value of date_fld3 after the move is 04/15/50.
C     *CDMY         MOVEL     char_fld5     date_fld3
 *  Date to character MOVEL (no separators)
 *  The result field length is longer than the factor 2 length.
 *  The value of char_fld4 after the move is '2010320XXX'.
C     *LONGJUL0     MOVEL     date_fld2     char_fld4
```

Figure 349. MOVEL between character and date fields

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL from variable to variable length
 * for character fields
 *
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
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL     var15a        var5a
 * var5a = 'FGHDE' (length=5)
C                   MOVEL     var10         var5b
 * var5b = '01234' (length=5)
C                   MOVEL     var5c         var15a
 * var15a = 'ABC' (length=3)
C                   MOVEL     var10         var15b
 * var15b = '012' (length=3)
```

Figure 350. MOVEL from a variable-length field to a variable-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL from variable to fixed length
 * for character fields
 *
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('')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL     var5          fix5a
 * fix5a = 'ABCDE'
C                   MOVEL     var10         fix5b
 * fix5b = '01234'
C                   MOVEL     var15         fix5c
 * fix5c = 'FGHPQ'
```

Figure 351. MOVEL from a variable-length field to fixed-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL from fixed to variable length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL     fix10         var5
 * var5 = 'PQRST' (length=5)
C                   MOVEL     fix10         var10
 * var10 = 'PQRSTUVWXY' (length=10)
C                   MOVEL     fix10         var15a
 * var15a = 'PQRSTUVWXYPQR' (length=13)
C                   MOVEL     fix10         var15b
 * var15b = 'PQRS' (length=4)
```

Figure 352. MOVEL from a fixed-length field to variable-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL(P) from variable to variable length
 * for character fields
 *
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
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL(P)  var15a        var5a
 * var5a = 'FGH  ' (length=5)
C                   MOVEL(P)  var10         var5b
 * var5b = '01234' (length=5)
C                   MOVEL(P)  var5c         var15b
 * var15b = 'ABC' (length=3)
C                   MOVEL(P)  var15a        var15c
 * var15c = 'FGH      ' (length=9)
```

Figure 353. MOVEL(P) from a variable-length field to a variable-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL(P) from variable to fixed length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL(P)  var5          fix5a
 * fix5a = 'ABCDE'
C                   MOVEL(P)  var10         fix5b
 * fix5b = '01234'
C                   MOVEL(P)  var15         fix5c
 * fix5c = 'FGH  '
```

Figure 354. MOVEL(P) from a variable-length field to fixed-length field

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
 *
 * Example of MOVEL(P) from fixed to variable length
 * for character fields
 *
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')
 *
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiL
 *
C                   MOVEL(P)  fix10         var5
 * var5 = 'PQRST' (length=5)
C                   MOVEL(P)  fix5          var10
 * var10 = '.....     ' (length=10)
C                   MOVEL(P)  fix10         var15a
 * var15a = 'PQRSTUVWXY   ' (length=13)
C                   MOVEL(P)  fix10         var15b
 * var15b = 'PQR' (length=3)
```

Figure 355. MOVEL(P) from a 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. Be sure that the result field is large enough to hold it. 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. 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 one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name. The result field must be numeric, but cannot be a named constant or literal. You can specify half adjust to have the result rounded.

For further information on the MULT operation, see “Arithmetic Operations” on page 434.

See Figure 172 on page 437 for 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. Factor 1 and factor 2 must be blank. The MVR operation must immediately follow the DIV operation. If you use conditioning indicators, ensure that the MVR operation is processed immediately after the DIV operation. If the MVR operation is processed before the DIV operation, undesirable results occur. The result field must be numeric and can contain one of: 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 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 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 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.

For further information on the MVR operation, see “Arithmetic Operations” on page 434.

See Figure 172 on page 437 for an example of the MVR operation.

NEXT (Next)

Free-Form Syntax	NEXT{(E)} *program-device file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
NEXT (E)	program-device	file-name		_	ER	_

The NEXT operation code forces the next input for a multiple device file to come from the program device specified by the *program-device* operand, providing the input operation is a cycle read or a READ-by-file-name. Any read operation, including CHAIN, EXFMT, READ, and READC, ends the effect of the previous NEXT operation. If NEXT is specified more than once between input operations, only the last operation is processed. The NEXT operation code can be used only for a multiple device file.

For the *program-device* operand, enter the name of a 10-character field that contains the program device name, a character literal, or named constant that is the program device name. The *file-name* operand is the name of the multiple device WORKSTN file for which the operation is requested.

To handle NEXT 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 79.

For more information, see “File Operations” on page 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
 * Assume devices Dev1 and Dev2  are connected to the WORKSTN file
 * DEVICEFILE. The first READ reads data from DEV1, the second READ
 * reads data from DEV2. The NEXT operation will direct the program
 * to wait for data from the device specified in factor 1 (i.e. DEV1)
 * for the third READ.
C
C                   READ (E)  Devicefile
C                   :
C                   READ (E)  Devicefile
C                   :
C     'DEV1'        NEXT
C                   :
C                   READ (E)  Devicefile
```

Figure 356. NEXT Operations

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 code specifies the occurrence of the data structure that is to be used next within an RPG IV program.

The OCCUR operation establishes which occurrence of a multiple occurrence data structure is used next in a program. Only one occurrence can be used at a time. 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.

Factor 1 is optional; if specified, it can contain a numeric, zero decimal position literal, field name, named constant, or a data structure name. Factor 1 is used during the OCCUR operation to set the occurrence of the data structure specified in factor 2. If factor 1 is blank, 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 is required and must be the name of a multiple occurrence data structure.

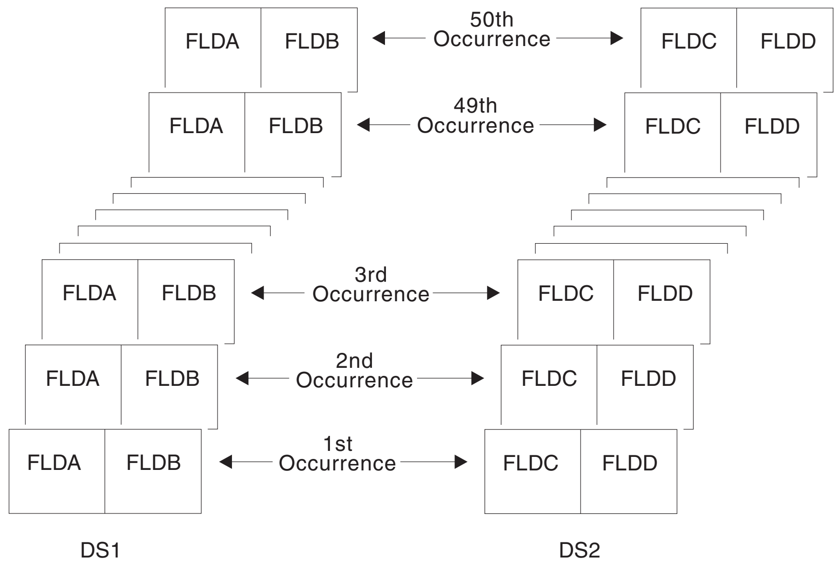
The result field is optional; if specified, it must be a numeric field name with no decimal positions. During the OCCUR operation, the value of the current occurrence of the data structure specified in factor 2, after being set by any value or data structure that is optionally specified in factor 1, is placed in the result field.

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/Errors” on page 96.

When a multiple-occurrence data structure is imported or exported, the information about the current occurrence is not imported or exported. See the “EXPORT{(external_name)}” on page 329 and “IMPORT{(external_name)}” on page 337 keywords for more information.




```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
 *
 *  DS1 and DS2 are multiple occurrence data structures.
 *  Each data structure has 50 occurrences.
D DS1             DS                    OCCURS(50)
D  FLDA                        1      5
D  FLDB                        6     80
 *
D DS2             DS                    OCCURS(50)
D  FLDC                        1      6
D  FLDD                        7     11
```

Figure 357. Uses of the OCCUR Operation (Part 1 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *  DS1 is set to the third occurrence.  The subfields FLDA
 *  and FLDB of the third occurrence can now be used.  The MOVE
 *  and Z-ADD operations change the contents of FLDA and FLDB,
 *  respectively, in the third occurrence of DS1.
C
C     3             OCCUR     DS1
C                   MOVE      'ABCDE'       FLDA
C                   Z-ADD     22            FLDB
 *
 *  DS1 is set to the fourth occurrence.  Using the values in
 *  FLDA and FLDB of the fourth occurrence of DS1, the MOVE
 *  operation places the contents of FLDA in the result field,
 *  FLDX, and the Z-ADD operation places the contents of FLDB
 *  in the result field, FLDY.
C
C     4             OCCUR     DS1
C                   MOVE      FLDA          FLDX
C                   Z-ADD     FLDB          FLDY
 *
 *  DS1 is set to the occurrence specified in field X.
 *  For example, if X = 10, DS1 is set to the tenth occurrence.
C     X             OCCUR     DS1
 *
 *  DS1 is set to the current occurrence of DS2.  For example, if
 *  the current occurrence of DS2 is the twelfth occurrence, DSI
 *  is set to the twelfth occurrence.
C     DS2           OCCUR     DS1
 *
 *  The value of the current occurrence of DS1 is placed in the
 *  result field, Z.  Field Z must be numeric with zero decimal
 *  positions.  For example, if the current occurrence of DS1
 *  is 15, field Z contains the value 15.
C                   OCCUR     DS1           Z
C
 *  DS1 is set to the current occurrence of DS2.  The value of the
 *  current occurrence of DS1 is then moved to the result field,
 *  Z.  For example, if the current occurrence of DS2 is the fifth
 *  occurrence, DS1 is set to the fifth occurrence.  The result
 *  field, Z, contains the value 5.
C
C     DS2           OCCUR     DS1           Z
 *
 *  DS1 is set to the current occurrence of X.  For example, if
 *  X = 15, DS1 is set to the fifteenth occurrence.
 *  If X is less than 1 or is greater than 50,
 *  an error occurs and %ERROR is set to return '1'.
 *  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 357. Uses of the OCCUR Operation (Part 2 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
 *
 * Procedure P1 exports a multiple occurrence data structure.
 * Since the information about the current occurrence is
 * not exported, P1 can communicate this information to
 * other procedures using parameters, but in this case it
 * communicates this information by exporting the current
 * occurrence.
 *
D EXP_DS          DS                   OCCURS(50) EXPORT
D  FLDA                        1      5
D NUM_OCCUR       C                    %ELEM(EXP_DS)
D EXP_DS_CUR      S               5P 0 EXPORT
 *
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
 *
 * Loop through the occurrences.  For each occurrence, call
 * procedure P2 to process the occurrence. Since the occurrence
 * number EXP_DS_CUR is exported, P2 will know which occurrence
 * to process.
 *
C                   DO        NUM_OCCUR     EXP_DS_CUR
C     EXP_DS_CUR    OCCUR     EXP_DS
C                     :
C                   CALLB     'P2'
C                   ENDDO
C                     :
```

Figure 358. Exporting a Multiple Occurrence DS (Part 1 of 2)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
 *
 * Procedure P2 imports the multiple occurrence data structure.
 * The current occurrence is also imported.
 *
D EXP_DS          DS                   OCCURS(50) IMPORT
D  FLDA                        1      5
D EXP_DS_CUR      S               5P 0 IMPORT
 *
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq.
 *
 * Set the imported multiple-occurrence data structure using
 * the imported current occurrence.
 *
C     EXP_DS_CUR    OCCUR     EXP_DS
 *
 * Process the current occurrence.
C                     :
```

Figure 358. Exporting a Multiple Occurrence DS (Part 2 of 2)

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 718.

For more information, see "Error-Handling Operations" on page 452.

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 named cannot be designated as a primary, secondary, or table file.

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 79.

To open the file specified in the *file-name* operand for the first time in a module or subprocedure with an explicit OPEN operation, specify the USROPN keyword on the file description specifications. (See Chapter 13, “File Description Specifications,” on page 279 for restrictions when using the USROPN keyword.)

If a file is opened and later closed by the CLOSE operation in the module or subprocedure, the programmer can reopen the file with the OPEN operation and the USROPN keyword on the file description specification is not required. When the USROPN keyword is not specified on the file description specification, the file is opened at module initialization for global files, or subprocedure initialization for local files. 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 is closed when the OPEN operation is issued to it.

When you open a file with the DEVID keyword specified (on the file description specifications), the fieldname specified as a parameter on the DEVID keyword is set to blanks. See the description of the DEVID keyword, in Chapter 13, “File Description Specifications,” on page 279.

For more information, see “File Operations” on page 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
F
FEXCEPTN   O    E             DISK    USROPN
FFILEX     F    E             DISK
F
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++..
 *
 *  The explicit OPEN operation opens the EXCEPTN file for
 *  processing if indicator 97 is on and indicator 98 is off.
 *  Note that the EXCEPTN file on the file description
 *  specifications has the USROPN keyword specified.
 *  %ERROR is set to return '1' if the OPEN operation fails.
 *
C                   IF        *in97 and not *in98
C                   OPEN(E)   EXCEPTN
C                   IF        not %ERROR
C                   WRITE     ERREC
C                   ENDIF
C                   ENDIF
 *
 *  FILEX is opened at program initialization.  The explicit
 *  CLOSE operation closes FILEX before control is passed to RTNX.
 *  RTNX or another program can open and use FILEX.  Upon return,
 *  the OPEN operation reopens the file.  Because the USROPN
 *  keyword is not specified for FILEX, the file is opened at
 *  program initialization
 *
C                   CLOSE     FILEX
C                   CALL      'RTNX'
C                   OPEN      FILEX
```

Figure 359. 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.

The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry must be the same as the entry for the associated DOUxx, DOWxx, IFxx, or WHENxx operation. 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. See "Compare Operations" on page 445.

Figure 307 on page 662 shows an example of ORxx and ANDxx operations with a DOUxx operation.

For more information, see "Structured Programming Operations" on page 469.

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 (When True Then Select)" on page 843 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.
- Within total calculations, the control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry is for documentation purposes only. Conditioning indicator entries (positions 9 through 11) are not allowed.

For more information, see "Structured Programming Operations" on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * Example of a SELECT group with WHENxx and OTHER.  If X equals 1,
 * do the operations in sequence 1; if X does not equal 1 and Y
 * equals 2, do the operations in sequence 2.  If neither
 * condition is true, do the operations in sequence 3.
 *
C                   SELECT
C     X             WHENEQ    1
 *
 * Sequence 1
 *
C                      :
C                      :
C     Y             WHENEQ    2
 *
 * Sequence 2
 *
C                      :
C                      :
C                   OTHER
 *
 * Sequence 3
 *
C                      :
C                      :
C                   ENDSL
```

Figure 360. OTHER Operation

For more details and examples, see the SELECT and WHENxx operations.

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. (The RPG IV language 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.

*LOCK cannot be specified when the *data-area-name* operand is the name of the local data area or the Program Initialization Parameters (PIP) data area.

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 IV exception/error handling routine receives control. If a message is issued to the requester, the message identifies the data area in error.

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/Errors" on page 96.

Positions 71-72 and 75-76 must be blank.

For further rules for the OUT operation, see "Data-Area Operations" on page 448.

See Figure 327 on page 702 for an example of the OUT operation.

PARM (Identify Parameters)

Free-Form Syntax	(not allowed - use “Prototypes and Parameters” on page 153 and CALLP)

Code	Factor 1	Factor 2	Result Field	Indicators		
PARM	Target field	Source field	Parameter			

The declarative PARM operation defines the parameters that compose a parameter list (PLIST). PARM operations can appear anywhere in calculations as long as they immediately follow the PLIST, CALL, or CALLB operation they refer to. PARM statements must be in the order expected by the called program or procedure. One PARM statement, or as many as 255 for a CALL or 399 for a CALLB or PLIST are allowed.

The PARM operation can be specified anywhere within calculations, including total calculations. The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement in the appropriate section of the program. Conditioning indicator entries (positions 9 through 11) are not allowed.

Factor 1 and factor 2 entries are optional. If specified, the entries must be the same type as specified in 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. A literal or named constant cannot be specified in factor 1. Factor 1 and factor 2 must be blank if the result field contains the name of a multiple-occurrence data structure or *OMIT.

TIP

If parameter type-checking is important for the application, you should define a prototype and procedure interface definition for the call interface, rather than use the PLIST and PARM operations.

The result field must contain the name of a:
- For all PARM statements:
  - Field
  - Data structure
  - Array
- For non-*ENTRY PLIST PARM statements it can also contain:
  - Array element
  - *OMIT (CALLB only)

The Result-Field entry of a PARM operation cannot contain:
- *IN, *INxx, *IN(xx)
- A literal
- A named constant
- A table name

In addition, the following are not allowed in the Result-Field entry of a PARM operation in the *ENTRY PLIST:
- *OMIT

- A globally initialized data structure
- A data structure with initialized subfields
- A data structure with a compile time array as a subfield
- Fields or data structures defined with the keywords BASED, IMPORT, or EXPORT
- An array element
- A data-area name
- A data-area data structure name
- A data-structure subfield
- A compile-time array
- A program status (PSDS) or file information data structure (INFDS)

A field name can be specified only once in an *ENTRY PLIST.

If an array is specified in the result field, the area defined for the array is passed to the called program or procedure. When a data structure with multiple occurrences is passed to the called program or procedure, 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.

Each parameter field has only one storage location; it is in the calling program or procedure. The address of the storage location of the result field is passed to the called program or procedure on a PARM operation. 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.

When a CALL or CALLB operation runs, the following occurs:
1. In the calling procedure, the contents of the factor 2 field of a PARM operation are copied into the result field (receiver field) of the same PARM operation.
2. In the case of a CALLB when the result field is *OMIT, a null address will be passed to the called procedure.
3. In the called procedure, after it 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 (receiver field) of the same PARM operation.
4. In the called procedure, when control is returned to the calling procedure, the contents of the factor 2 field of a PARM operation are copied into the result

field (receiver field) of the same PARM operation. This move does not occur if the called procedure ends abnormally. The result of the move is unpredictable if an error occurs on the move.

5. Upon return to the calling procedure, the contents of the result field of a PARM operation in the calling procedure are copied into the factor 1 field (receiver field) of the same PARM operation. This move does not occur if the called procedure ends abnormally or if an error occurs on the call operation.

Note: The data is moved in the same way as data is moved using the EVAL operation code. Strict type compatibility is enforced. For a discussion of how to call and pass parameters to a program through CL, see the *CL Programming* manual.

For more information, see “Call Operations” on page 440 or “Declarative Operations” on page 452.

Figure 361 on page 769 illustrates the PARM operation.

PLIST (Identify a Parameter List)

Free-Form Syntax	(not allowed - use “Prototypes and Parameters” on page 153 and CALLP)

Code	Factor 1	Factor 2	Result Field	Indicators		
PLIST	PLIST name					

The declarative PLIST operation defines a unique symbolic name for a parameter list to be specified in a CALL or CALLB operation.

You can specify a PLIST operation anywhere within calculations, including within total calculations and between subroutines. The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement in the appropriate section of the program. The PLIST operation must be immediately followed by at least one PARM operation. Conditioning indicator entries (positions 9 through 11) are not allowed.

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 procedure. A parameter list is ended when an operation other than PARM is encountered.

TIP

If parameter type-checking is important for the application, you should define a prototype and procedure inter- face definition for the call interface, rather than use the PLIST and PARM operations.

For more information, see “Call Operations” on page 440 or “Declarative Operations” on page 452.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  In the calling program, the CALL operation calls PROG1 and
 *  allows PROG1 to access the data in the parameter list fields.
C                   CALL     'PROG1'        PLIST1
 *
 *  In the second PARM statement, when CALL is processed, the
 *  contents of factor 2, *IN27, are placed in the result field,
 *  BYTE.  When PROG1 returns control, the contents of the result
 *  field, BYTE, are placed in the factor 1 field, *IN30.  Note
 *  that factor 1 and factor 2 entries on a PARM are optional.
 *
C     PLIST1        PLIST
C                   PARM                    Amount            5 2
C     *IN30         PARM      *IN27         Byte              1
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   CALLB     'PROG2'
 *  In this example, the PARM operations immediately follow a
 *  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....+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *  In the called procedure, PROG2,  *ENTRY in factor 1 of the
 *  PLIST statement identifies it as the entry parameter list.
 *  When control transfers to PROG2, the contents of the result
 *  fields (FieldC and FieldG) of the parameter list are placed in
 *  the factor 1 fields (FieldA and FieldD).  When the called procedure
 *  returns, the contents of the factor 2 fields of the parameter
 *  list (FieldB and FieldE) are placed in the result fields (FieldC
 *  and FieldG).  All of the fields are defined elsewhere in the
 *  procedure.
C     *ENTRY        PLIST
C     FieldA        PARM      FieldB        FieldC
C     FieldD        PARM      FieldE        FieldG
```

Figure 361. PLIST/PARM Operations

POST (Post)

Free-Form Syntax	POST{(E)} {*program-device*} *file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
POST (E)	program-device	file-name	INFDS name	_	ER	_

The POST operation puts information in an INFDS (file information data structure). This information contains the following:

- File Feedback Information specific to RPG I/O for the file
- Open Feedback Information for the file
- Input/Output Feedback Information and Device Dependent Feedback Information for the file OR Get Attribute Information

The *program-device* operand specifies a program device name to get information about that specific program device. If you specify a program device, the file must be defined as a WORKSTN file. If *program-device* is specified, then the INFDS will contain Get Attribute Information following the Open Feedback Information. Use either a character field of length 10 or less, a character literal, or a character named constant. If *program-device* is not specified, then the INFDS will contain Input/Output Feedback Information and Device Dependent Feedback Information following the Open 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.

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 79.

Even when a POST operation code is not processed, its existence in your program can affect the way the RPG IV language operates. The presence of a POST operation with no program-device specified can affect the posting of feedback to one or more files.

- The presence of a POST operation with no program-device specified for a file defined on a global File specification will affect the implicit posting of feedback to the INFDS for all global files in the module.
- The presence of a POST operation with no program-device specified for a global file will have no effect on the implicit posting of feedback to the INFDS for files defined in subprocedures.

- The presence of a POST operation with no program-device specified for a locally-defined file will only affect the implicit posting of feedback to the INFDS for that file; it will have no effect for global files, or for other files defined in that subprocedure.
- The implicit posting of feedback to the INFDS for a file that is passed as a parameter is determined by the module in which the file is defined. A POST operation with no program-device specified to a file parameter may be redundant if the feedback information is always posted to that file's INFDS.
- If a global file is passed as a parameter to another procedure in the same module, and that procedure does a POST operation to its parameter, that POST operation will not be considered to be a POST operation to a global file.

Usually, the INFDS is updated at each input and output operation or block of operations. However, if the presence of a POST operation affects the posting of feedback to the INFDS of a file, then RPG IV updates the I/O Feedback Information area and the Device Dependent Feedback Information area in the INFDS of the file only when you process a POST operation for the file. The File Dependent Information in the INFDS is updated on all Input/Output operations. If you have opened a file for multiple-member processing, the Open Feedback Information in the INFDS will be updated when an input operation (READ, READP, READE READPE) causes a new member to be opened.

Note that DUMP retrieves its information directly from the Open Data Path and not from the INFDS, so the file information sections of the DUMP do not depend on POST.

If a program has no POST operation code, or if it has only POST operation codes with *program-device* specified, the Input/Output Feedback and Device Dependent Feedback section is updated with each input/output operation or block of operations. If RPG is blocking records, most of the information in the INFDS will be valid only for the last complete block of records processed. When doing blocked input, from a data base file, RPG will update the relative record number and key information in the INFDS for each read, not just the last block of records processed. If you require more accurate information, do not use record blocking. See “File Information Data Structure” on page 79 for more information on record blocking. If you do not require feedback information after every input/output operation, you may be able to improve performance by using the POST operation only when you require the feedback information.

When a POST operation is processed, the associated file must be open. If you specify a program device on the POST operation, it does not have to be acquired by the file.

For more information, see “File Operations” on page 453.

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

The READ operation reads the record, currently pointed to, from a full procedural file (identified by an F in position 18 of the file description specifications).

The *name* operand is required and must be the name of a 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 453 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 a record-lock error (status 1218), the file is still positioned at the locked record and the next read operation will attempt to read that record again. Otherwise, if there is any other error or an end of file condition, you must reposition the file (using a CHAIN, SETLL, or SETGT operation).

If the file from which you are reading is an update disk file, you can specify an N operation extender to indicate that no lock should be placed on the record when it is read. See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

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 79.

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.

Figure 362 on page 774 illustrates the READ operation.

When *name* specifies a multiple device file, the READ operation does one of the following:

- Reads data from the device specified in the most recent NEXT operation (if such a NEXT operation has been processed).
- Accepts the first response from any device that has been acquired for the file, and that was specified for "invite status" with the DDS keyword INVITE. If there are no invited devices, the operation receives an end of file. The input is processed according to the corresponding format. If the device is a workstation, the last format written to it is used. If the device is a communications device, you can select the format.

  Refer to *ICF Programming*, SC41-5442-00 for more information on format selection processing for an ICF file.

  The READ operation will stop waiting after a period of time in which no input is provided, or when one of the following CL commands has been entered with the controlled option specified:
  - ENDJOB (End Job)
  - ENDSBS (End Subsystem)
  - PWRDWNSYS (Power Down System)
  - ENDSYS (End System).

  This results in a file exception/error that is handled by the method specified in your program (see "File Exception/Errors" on page 79). See *ICF Programming*, SC41-5442-00 for a discussion of the WAITRCD parameter on the commands to create or modify a file. This parameter controls the length of time the READ operation waits for input.

When *name* specifies a format name and the format name is associated with a multiple device file, data is read from the device identified by the field specified in the DEVID keyword in file specifications. If there is no such entry, data is read from the device used in the last successful input operation.

See "Database Null Value Support" on page 219 for information on reading records with null-capable fields.

For more information, see "File Operations" on page 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  READ retrieves the next record from the file FILEA, which must
 *  be a full procedural file.
 *  %EOF is set to return '1' if an end of file occurs on READ,
 *  or if an end of file has occurred previously and the file
 *  has not been repositioned.  When %EOF returns '1',
 *  the program will leave the loop.
 *
C                   DOW       '1'
C                   READ      FILEA
C                   IF        %EOF
C                   LEAVE
C                   ENDIF
 *
 *  READ retrieves the next record of the type REC1 (factor 2)
 *  from an externally described file.  (REC1 is a record format
 *  name.)  Indicator 64 is set on if an end of file occurs on READ,
 *  or if it has occurred previously and the file has not been
 *  repositioned.  When indicator 64 is set on, the program
 *  will leave the loop.  The N operation code extender
 *  indicates that the record is not locked.
 *
C                   READ(N)   REC1                                   64
C   64              LEAVE
C                   ENDDO
```

Figure 362. READ Operation

READC (Read Next Changed Record)

Free-Form Syntax	READC{(E)} *record-name {data-structure}*

Code	Factor 1	Factor 2	Result Field	Indicators		
READC (E)		record-name	data structure	_	ER	EOF

The READC operation can be used only with an externally described WORKSTN file to obtain the next changed record in a subfile. The *record-name* operand is required and must be the name of a record format defined as a subfile by the SFILE keyword on the file description specifications. (See “SFILE(recformat:rrnfield)” on page 309 for information on the SFILE keyword.)

For a multiple device file, data is read from the subfile record associated with a program device; the program device is identified by the field specified in the DEVID keyword on the file specifications. If there is no such entry, data is read from the program device used for the last successful input operation.

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 79.

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 the *data-structure* operand is specified, the record is read directly into the data structure. The data structure must be a data structure defined with EXTNAME(...:*INPUT) or LIKEREC(...:*INPUT). See “File Operations” on page 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
 * CUSSCR is a WORKSTN file which displays a list of records from
 * the CUSINFO file. SFCUSR is the subfile name.
 *
FCUSINFO   UF   E             DISK
FCUSSCR    CF   E             WORKSTN SFILE(SFCUSR:RRN)
F
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * After the subfile has been loaded with the records from the
 * CUSINFO file. It is written out to the screen using EXFMT with
 * the subfile control record, CTLCUS. If there are any changes in
 * any one of the records listed on the screen, the READC operation
 * will read the changed records one by one in the do while loop.
 * The corresponding record in the CUSINFO file will be located
 * with the CHAIN operation and will be updated with the changed
 * field.
C                   :
C                   EXFMT     CTLCUS
C                   :
 * SCUSNO, SCUSNAM, SCUSADR, and SCUSTEL are fields defined in the
 * subfile. CUSNAM, CUSADR, and CUSTEL are fields defined in a
 * record, CUSREC which is defined in the file CUSINFO.
 *
C                   READC     SFCUSR
C                   DOW       %EOF = *OFF
C     SCUSNO        CHAIN (E) CUSINFO
 * 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) SFCUSR
C                   ENDDO
```

Figure 363. 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 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 289 on page 635 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 546 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*.

Note: Note: If a file is defined as update and the N operation extender is not specified, occasionally a READE operation will be forced to wait for a temporary record lock for a record whose key value does not match the search argument. Once the temporary lock has been obtained, if the key value does not match the search argument, the temporary lock is released.

In most cases, RPG can perform READE by using system support that does not require obtaining a temporary record lock to determine whether there is a matching record. However, in other cases, RPG cannot use this support, and must request the next record before it can determine whether the record matches the READE request.

Some of the reasons that would require RPG to obtain a temporary lock on the next record for a READE operation are:

- the key of the current record is not the same as the search argument
- the current record is not the same as the requested record
- there are null-capable fields in the file
- the file has end-of-file delay

Note:

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 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

If the file you are reading is an update disk file, you can specify an N operation extender to indicate that no lock should be placed on the record when it is read. See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

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 79.

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 a READE operation is successful, the file is positioned at the next record that satisfies the operation. If there is a record-lock error (status 1218), the file is still positioned at the locked record and the next read operation will attempt to read that record again. Otherwise, if there is any other error or an end of file condition, you must reposition the file (using a CHAIN, SETLL, or SETGT operation). See “CHAIN (Random Retrieval from a File)” on page 633, “SETGT (Set Greater Than)” on page 804, or “SETLL (Set Lower Limit)” on page 808.

Normally, the comparison between the specified key and the actual key in the file is done by data management. In some cases this is impossible, causing the comparison to be done using the hexadecimal collating sequence. This can give different results than expected. For more information, see the section "Unexpected Results Using Keyed Files" in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

A READE with the *search-arg* operand 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 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. 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 219 for information on handling records with null-capable fields and keys.

For more information, see “File Operations” on page 453.

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...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  With Factor 1 Specified...
 *
 *  The READE operation retrieves the next record from the file
 *  FILEA and compares its key to the search argument, KEYFLD.
 *
 *  The %EOF built-in function is set to return '1' if KEYFLD is
 *  not equal to the key of the record read or if end of file
 *  is encountered.
 *
C     KEYFLD        READE     FILEA
 *
 *  The READE operation retrieves the next record of the type REC1
 *  from an externally described file and compares the key of the
 *  record read to the search argument, KEYFLD.  (REC1 is a record
 *  format name.)  Indicator 56 is set on if KEYFLD is not equal to
 *  the key of the record read or if end of file is encountered.
C     KEYFLD        READE     REC1                                   56
 *
 *  With No Factor 1 Specified...
 *
 *  The READE operation retrieves the next record in the access
 *  path from the file FILEA if the key value is equal to
 *  the key value of the record at the current cursor position.
 *
 *  If the key values are not equal, %EOF is set to return '1'.
C                   READE     FILEA
 *
 *  The READE operation retrieves the next record in the access
 *  path from the file FILEA if the key value equals the key value
 *  of the record at the current position.  REC1 is a record format
 *  name.  Indicator 56 is set on if the key values are unequal.
 *  N indicates that the record is not locked.
C                   READE(N)  REC1                                   56
```

Figure 364. 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 in *name*, 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 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

If a READP operation is successful, the file is positioned at the previous record that satisfies the read.

If the file from which you are reading is an update disk file, you can specify an N operation extender to indicate that no lock should be placed on the record when it is read. See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

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 79.

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.

If there is a record-lock error (status 1218), the file is still positioned at the locked
record and the next read operation will attempt to read that record again.
Otherwise, if there is any other error or a beginning of file condition, you must
reposition the file (using a CHAIN, SETLL, or SETGT operation).

See "Database Null Value Support" on page 219 for information on reading records with null-capable fields.

For more information, see "File Operations" on page 453.

Figure 365 shows READP operations with a file name and record format name specified in factor 2.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The READP operation reads the prior record from FILEA.
 *
 *  The %EOF built-in function is set to return '1' if beginning
 *  of file is encountered.  When %EOF returns '1', the program
 *  branches to the label BOF specified in the GOTO operation.
C                   READP     FILEA
C                   IF        %EOF
C                   GOTO      BOF
C                   ENDIF
 *
 *  The READP operation reads the next prior record of the type
 *  REC1 from an externally described file.  (REC1 is a record
 *  format name.)  Indicator 72 is set on if beginning of file is
 *  encountered during processing of the READP operation.  When
 *  indicator 72 is set on, the program branches to the label BOF
 *  specified in the GOTO operation.
C                   READP     PREC1                                  72
C   72              GOTO      BOF
 *
C     BOF           TAG
```

Figure 365. 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 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 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 289 on page 635 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 546 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 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

If the file from which you are reading is an update disk file, you can specify an N operation extender to indicate that no lock should be placed on the record when it is read. See the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

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 79.

You can specify an indicator in positions 75-76 that will be set on if a BOF 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 there is a record-lock error (status 1218), the file is still positioned at the locked record and the next read operation will attempt to read that record again. Otherwise, if there is any other error or a beginning of file condition, you must reposition the file (using a CHAIN, SETLL, or SETGT operation). See “CHAIN (Random Retrieval from a File)” on page 633, “SETGT (Set Greater Than)” on page 804, or “SETLL (Set Lower Limit)” on page 808.

Note: Note: If a file is defined as update and the N operation extender is not specified , occasionally a READPE operation will be forced to wait for a temporary record lock for a record whose key value does not match the search argument. Once the temporary lock has been obtained, if the key value does not match the search argument, the temporary lock is released.

In most cases, RPG can perform READPE by using system support that does not require obtaining a temporary record lock to determine whether there is a matching record. However, in other cases, RPG cannot use this support, and must request the next record before it can determine whether the record matches the READPE request.

Some of the reasons that would require RPG to obtain a temporary lock on the next record for a READPE operation are:

- the key of the current record is not the same as the search argument
- the current record is not the same as the requested record
- there are null-capable fields in the file
- the file has end-of-file delay

Normally, the comparison between the specified key and the actual key in the file is done by data management. In some cases this is impossible, causing the comparison to be done using the hexadecimal collating sequence. This can give different results than expected. For more information, see the section "Unexpected Results Using Keyed Files" in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

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.

See “Database Null Value Support” on page 219 for information on handling records with null-capable fields and keys.

For more information, see "File Operations" on page 453.

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...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * With Factor 1 Specified...
 *
 * The previous record is read and the key compared to FieldA.
 * Indicator 99 is set on if the record's key does not match
 * FieldA.
C     FieldA        READPE    FileA                                  99
 *
 * The previous record is read from FileB and the key compared
 * to FieldB.  The record is placed in data structure Ds1.  If
 * the record key does not match FieldB, indicator 99 is set on.
C     FieldB        READPE    FileB         Ds1                      99
 *
 * The previous record from record format RecA is read, and
 * the key compared to FieldC.  Indicator 88 is set on if the
 * operation is not completed successfully, and 99 is set on if
 * the record key does not match FieldC.
C     FieldC        READPE    RecA                                 8899
 *
 * With No Factor 1 Specified...
 *
 * The previous record in the access path is retrieved if its
 * key value equals the key value of the current record.
 * Indicator 99 is set on if the key values are not equal.
C                   READPE    FileA                                  99
 *
 * The previous record is retrieved from FileB if its key value
 * matches the key value of the record at the current position
 * in the file.  The record is placed in data structure Ds1.
 * Indicator 99 is set on if the key values are not equal.
C                   READPE    FileB         Ds1                      99
 *
 * The previous record from record format RecA is retrieved if
 * its key value matches the key value of the current record in
 * the access path.  Indicator 88 is set on if the operation is
 * not successful; 99 is set on if the key values are unequal.
C                   READPE    RecA                                 8899
```

Figure 366. READPE 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 such as CEEGTST). 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 426 (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/Errors” on page 96.

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 the maximum size allowed.

The maximum size allowed depends on the type of heap storage used for memory management operations due to the ALLOC keyword on the Control specification. If it is known at compile time that the module uses the teraspace storage model for memory management operations, the maximum size allowed is 4294967295 bytes. Otherwise, the maximum size allowed is 16776704 bytes.

The maximum size available at runtime may be less than the maximum size allowed by RPG.

When RPG memory management operations for the module are using single-level heap storage due to the ALLOC keyword on the Control specification, the REALLOC operation can only handle pointers to single-level heap storage. When RPG memory management operations for the module are using teraspace heap storage, the REALLOC operation can handle pointers to both single-level and teraspace heap storage.

For more information, see “Memory Management Operations” on page 458.

```
D Ptr1            S                *
D Fld             S           32767A     BASED(Ptr1)
 * The ALLOC operation allocates 7 bytes to the pointer Ptr1.
 * 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                   REALLOC   10                Ptr1
 * Now 10 bytes of Fld can be used.
C                   EVAL      %SUBST(Fld : 1 : 10) = '123456789A'
```

Figure 367. REALLOC Operation

REL (Release)

Free-Form Syntax	REL{(E)} *program-device file-name*

Code	Factor 1	Factor 2	Result Field	Indicators		
REL (E)	program-device	file-name		_	ER	_

The REL operation releases the program device specified in *program-device* from the WORKSTN file specified in *file-name*.

Specify the program device name in the *program-device* operand. Use either a character field of length 10 or less, a character literal, or a named constant. Specify the file name in *file-name* operand.

To handle REL 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 79.

When there are no program devices acquired to a WORKSTN file, the next READ-by-file-name or cycle-read gets an end-of-file condition. You must decide what the program does next. The REL operation may be used with a multiple device file or, for error recovery purpose, with a single device file.

Note: To release a record lock, use the UNLOCK operation. See the UNLOCK operation for more information about releasing record locks for update disk files.

For more information, see “File Operations” on page 453.

RESET (Reset)

Free-Form Syntax	RESET{(E)} {*NOKEY} {*ALL} *name*

Code	Factor 1	Factor 2	Result Field	Indicators		
RESET (E)	*NOKEY	*ALL	name (variable or record format)	_	ER	_

The RESET operation is used to restore a variable to the value held at the end of the *INIT phase. This value is called the **reset value**. If there is no *INZSR subroutine, the reset value is the same as the initial value (either the value specified by the "INZ{(initial value)}" on page 338, or the default value). If there is a *INZSR subroutine, the reset value is the value the variable holds when the *INZSR subroutine has completed.

The RESET operation can also be used to restore all the fields in a record format to their reset values.

See Figure 8 on page 33 for more information on the *INIT phase.

Note: For local variables in subprocedures, the reset value is the value of the variable when the subprocedure is first called, but before the calculations begin.

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/Errors" on page 96.

For more information, see "Initialization Operations" on page 457.

Resetting Variables

*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 reset and the occurrence level or table index is set to 1.

The *name* operand specifies the variable to be reset. The particular value for this operand determines the reset action as follows:

Single occurrence data structure
All fields are reset in the order in which they are declared within the structure.

Multiple-occurrence data structure
If *ALL is not specified, then all fields in the *current* occurrence are reset. If *ALL is specified, then all fields in *all* occurrences are reset.

Table name
If *ALL is not specified, then the *current* table element is reset. If *ALL is specified, then all table elements are reset.

Array name
Entire array is reset

Array element (including indicators)
Only the element specified is reset.

Resetting Record Formats

*NOKEY is optional. If *NOKEY is specified, then key fields are not reset to their reset values.

*ALL is optional. If *ALL is specified and *NOKEY is not, all fields in the record format are reset. 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 reset, even if *ALL is specified.

The result field contains the record format to be reset. For WORKSTN file record formats (positions 36-42 on a file-description specification), if *ALL is not specified, only those fields with a usage of output or both are affected. All field-conditioning indicators of the record format are affected by the operation. When the RESET operation is applied to a record format name, and INDARA has been specified in the DDS, the indicators in the record format are not reset.

Fields in DISK, SEQ, 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.

A RESET operation of a record format with *ALL specified is not valid when:

- A field is defined externally as input-only, and the record was not used for input.
- A field is defined externally as output-only, and the record was not used for output.
- A field is defined externally as both input and output capable, and the record was not used for either input or output.

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 *ALL specified is done to a record containing these fields, then these fields will be cleared or reset because they appear in the output specifications.

Additional Considerations

Keep in mind the following when coding a RESET operation:

- RESET is not allowed for based variables and IMPORTed variables, or for parameters in a subprocedure.
- The RESET operation results in an increase in the amount of storage required by the program. For any variable that is reset, the storage requirement is doubled. Note that for multiple occurrence data structures, tables and arrays, the reset value of every occurrence or element is saved.
- If a RESET occurs during the initialization routine of the program, an error message will be issued at run time. If a GOTO or CABxx is used to leave subroutine calculations during processing of the *INZSR, or if control passes to another part of the cycle as the result of error processing, the part of the initialization step which initializes the save areas will never be reached. In this case, an error message will be issued for all RESET operations in the program at run time.
- A RESET operation within a subprocedure to a global variable or structure is valid in the following circumstances:
  - If there is no *INZSR, it is always valid

  - If there is a *INZSR, it is not valid until the *INZSR has completed at least once. After that, it is always valid, even if the cycle-main procedure is not active.
- Performing a RESET operation on a parameter of a *ENTRY PLIST that does not get passed when the program is called may cause unpredictable results. An alternative would be to save the parameter value into a variable defined LIKE the parameter if the value returned by %PARMS() indicates that the parameter is passed.

Attention!

When the RESET values are saved, a pointer-not-set error will occur if the following are *all* true in a cycle module:

- There is no *INZSR
- An entry parameter to the cycle-main procedure is RESET anywhere in the module
- A subprocedure is called before the cycle-main procedure has ever been called

For more information, see “CLEAR (Clear)” on page 642.

RESET Examples

Except for the actual operation performed on the fields, the considerations shown in the following examples also apply to the CLEAR operation. Figure 368 on page 791 shows an example of the RESET operation with *NOKEY.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
EXTFILE   O    E             DISK
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
 * The file EXTFILE contains one record format RECFMT containing
 * the character fields CHAR1 and CHAR2 and the numeric fields
 * NUM1 and NUM2.  It has keyfields CHAR2 and NUM1.
D
D DS1             DS
D  DAY1                   1      8    INZ('MONDAY')
D  DAY2                   9     16    INZ('THURSDAY')
D  JDATE                 17     22
D
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq...
 *
 * The following operation sets DAY1, DAY2, and JDATE to blanks.
C
C                   CLEAR                   DS1
C
 * The following operation will set DAY1, DAY2, and JDATE to their
 * reset values of 'MONDAY', 'THURSDAY', and UDATE respectively.
 * The reset value of UDATE for JDATE is set in the *INZSR.
C
C                   RESET                   DS1
C
 * The following operation will set CHAR1 and CHAR2 to blanks and
 * NUM1 and NUM2 to zero.
C                   CLEAR                   RECFMT
 * The following operation will set CHAR1, CHAR2, NUM1, and
 * NUM2 to their reset values of 'NAME', 'ADDRESS', 1, and 2
 * respectively. These reset values are set in the *INZSR.
 *
C                   RESET                   RECFMT
 * The following operation sets all fields in the record format
 * to blanks, except the key fields CHAR2 and NUM1.
 *
C     *NOKEY        RESET     *ALL          RECFMT
C                   RETURN
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
ORCDNAME+++D...N01N02N03EXCNAM++++.......................................
O..............N01N02N03FIELD+++++++++.B.................................
ORECFMT    T
O                       CHAR1
O                       CHAR2
O                       NUM1
O                       NUM2
```

*Figure 368. RESET Operation with *NOKEY*

```
     A          R RECFMT
     A            CHAR1         10A
     A            CHAR2         10A
     A            NUM1           5P 0
     A            NUM2           7S 2
```

Figure 369. DDS for EXTFILE

Figure 370 on page 793 shows an excerpt of a source listing for a program that uses two externally described files, RESETIB and RESETON. Each has two record formats, and each record format contains an input field FLDIN, an output field FLDOUT, and a field FLDBOTH, that is input-output capable. The DDS are shown in Figure 371 on page 794 and Figure 372 on page 794.

Because RESETIB is defined as a combined file, the fields for RECBOTH, which are defined as input-output capable, are available on both input and output specifications. On the other hand, the fields for RECIN are on input specifications only.

```
 1  * The file RESETIB contains 2 record formats RECIN and RECBOTH.
 2 FRESETIB   CF   E             WORKSTN
 3  * The file RESETON contains 2 record formats RECOUT and RECNONE.
 4 FRESETON   O    E             WORKSTN
 5
 6=IRECIN
 7=I                             A    1    1  *IN02
 8=I                             A    2   11  FLDIN
 9=I                             A   12   21  FLDBOTH
10=IRECBOTH
11=I                             A    1    1  *IN04
12=I                             A    2   11  FLDIN
13=I                             A   12   21  FLDBOTH
14 C                   WRITE     RECOUT
15 C                   WRITE     RECBOTH
16 C                   READ      RECIN                              ----99
17 C                   READ      RECBOTH                            ----99
18
19  * RESET without factor 2 means to reset only those fields which
20  * appear on the output specifications for the record format.
21  * Since only RECOUT and RECBOTH have write operations, the
22  * RESET operations for RECNONE and RECIN will have no effect.
23  * The RESET operations for RECOUT and RECBOTH will reset fields
24  * FLDOUT and FLDBOTH.  FLDIN will not be affected.
25 C                   RESET                   RECNONE
26 C                   RESET                   RECIN
27 C                   RESET                   RECOUT
28 C                   RESET                   RECBOTH
29
30  * RESET with *ALL in factor 2 means to reset all fields.  Note
31  * that this can only be done when all fields are used in at least
32  * one of the ways they are defined (for example, an output-capable
33  * field must be used for output by the record format)
34  * Since RECNONE does not have either input or output operations,
35  * the RESET *ALL for RECNONE will fail at compile time.
36  * Since RECIN does not have any output operations, RESET *ALL RECIN
37  * will fail because FLDOUT is not output.
38  * Since RECOUT does not have any input operations, and is not defined
39  * as input capable on the file specification, RESET *ALL RECOUT
40  * will fail because FLDIN is not input.
41  * The RESET *ALL for RECBOTH will reset all fields: FLDIN, FLDOUT
42  * and FLDBOTH.
43 C                   RESET     *ALL          RECNONE
44 C                   RESET     *ALL          RECIN
45 C                   RESET     *ALL          RECOUT
46 C                   RESET     *ALL          RECBOTH
47
48 C                   SETON                                        LR----
49=ORECBOTH
50=O                       *IN14                1A CHAR        1
51=O                       FLDOUT              11A CHAR       10
52=O                       FLDBOTH             21A CHAR       10
53=ORECOUT
54=O                       *IN13                1A CHAR        1
55=O                       FLDOUT              11A CHAR       10
56=O                       FLDBOTH             21A CHAR       10
```

*Figure 370. RESET with *ALL – Source Listing Excerpt.* The input and output specifications with '=' after the listing line number are generated by the compiler.

When the source is compiled, several errors are identified. Both RECNONE and RECIN are identified as having no output fields. The RESET *ALL is disallowed for all but the RECBOTH record, since it is the only record format for which all fields appear on either input or output specifications.

```
     A          R RECIN                       CF02(02)
     A            FLDIN         10A  I  2  2
     A            FLDOUT        10A  O  3  2
     A  12        FLDBOTH       10A  B  4  2
     A          R RECBOTH                     CF04(04)
     A            FLDIN         10A  I  2  2
     A            FLDOUT        10A  O  3  2
     A  14        FLDBOTH       10A  B  4  2
```

Figure 371. DDS for RESETIB

```
     A          R RECNONE                     CF01(01)
     A            FLDIN         10A  I  2  2
     A            FLDOUT        10A  O  3  2
     A  11        FLDBOTH       10A  B  4  2
     A          R RECOUT                      CF03(03)
     A            FLDIN         10A  I  2  2
     A            FLDOUT        10A  O  3  2
     A  13        FLDBOTH       10A  B  4  2
```

Figure 372. DDS for RESETON

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 a value is returned to the caller, the return value is specified in the *expression* operand.

The actions which occur as a result of the RETURN operation differ depending on whether the operation is in a cycle-main procedure or subprocedure. When a cycle-main procedure returns, the following occurs:

1. The halt indicators are checked. If a halt indicator is on, the procedure ends abnormally. (All open files are closed, an error return code is set to indicate to the calling routine that the procedure has ended abnormally, and control returns to the calling routine.)
2. If no halt indicators are on, the LR indicator is checked. If LR is on, the program ends normally. (Locked data area structures, arrays, and tables are written, and external indicators are reset.)
3. If no halt indicator is on and LR is not on, the procedure returns to the calling routine. Data is preserved for the next time the procedure is run. Files and data areas are not written out. See the chapter on calling programs and procedures in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for information on how running in a *NEW activation group affects the operation of RETURN.

When a subprocedure returns, the return value, if specified on the prototype of the called program or procedure, is passed to the caller. Automatic files are closed. Nothing else occurs automatically. All static or global files and data areas must be closed manually. You can set on indicators such as LR, 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 486.

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 extended factor 2 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.

Attention!

If the subprocedure 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.

Performance tip

Specifying the RTNPARM keyword on your prototype may significantly improve the performance for returning large values. See "RTNPARM" on page 363 for more information.

For more information, see "Call Operations" on page 440.

```
 * This is the prototype for subprocedure RETNONE. Since the
 * prototype specification does not have a data type, this
 * subprocedure does not return a value.
D RetNone         PR
 * This is the prototype for subprocedure RETFLD. Since the
 * prototype specification has the type 5P 2, this subprocedure
 * returns a packed value with 5 digits and 2 decimals.
 * The subprocedure has a 5-digit integer parameter, PARM,
 * passed by reference.
D RetFld          PR             5P 2
D   Parm                         5I 0
 * This is the prototype for subprocedure RETARR. The data
 * type entries for the prototype specification show that
 * this subprocedure returns a date array with 3 elements.
 * The dates are in *YMD/ format.
D RetArr          PR              D   DIM(3) DATFMT(*YMD/)
 * This procedure (P) specification indicates the beginning of
 * subprocedure RETNONE. The data specification (D) specification
 * immediately following is the procedure-interface
 * specification for this subprocedure. Note that the
 * procedure interface is the same as the prototype except for
 * the definition type (PI vs PR).
P RetNone         B
D RetNone         PI
 * RetNone does not return a value, so the RETURN
 * operation does not have factor 2 specified.
C                   RETURN
P RetNone         E
 * The following 3 specifications contain the beginning of
 * the subprocedure RETFLD as well as its procedure interface.
P RetFld          B
D RetFld          PI             5P 2
D   Parm                         5I 0
D Fld             S             12S 1 INZ(13.8)
 * RetFld returns a numeric value.  The following RETURN
 * operations show returning a literal, an expression and a
 * variable. Note that the variable is not exactly the same
 * format or length as the actual return value.
C                   RETURN       7
C                   RETURN       Parm * 15
C                   RETURN       Fld
P RetFld          E
```

Figure 373. Examples of the RETURN Operation (Part 1 of 2)

```
 * The following 3 specifications contain the beginning of the
 * subprocedure RETARR as well as its procedure interface.
P RetArr          B
D RetArr          PI              D   DIM(3)
D SmallArr        S               D   DIM(2) DATFMT(*ISO)
D BigArr          S               D   DIM(4) DATFMT(*USA)
 * RetArr returns a date array.  Note that the date
 * format of the value specified on the RETURN operation
 * does not have to be the same as the defined return
 * value.
 * The following RETURN operation specifies a literal.
 * The caller receives an array with the value of the
 * literal in every element of the array.
C                   RETURN        D'1995-06-27'
 * The following return operation returns an array
 * with a smaller dimension than the actual return value.
 * In this case, the third element would be set to the
 * default value for the array.
C                   RETURN        SmallArr
 * The following return operation returns an array
 * with a larger dimension than the actual return
 * value.  In this case, the fourth element of BigArr
 * would be ignored.
C                   RETURN        BigArr
P RetArr          E
```

Figure 373. Examples of the RETURN Operation (Part 2 of 2)

ROLBK (Roll Back)

Free-Form Syntax	ROLBK{(E)}

Code	Factor 1	Factor 2	Result Field	Indicators		
ROLBK (E)				_	ER	_

The ROLBK operation:
- Eliminates all the changes to your files that have been specified in output operations since the previous COMMIT or ROLBK operation (or since the beginning of operations under commitment control if there has been no previous COMMIT or ROLBK operation).
- Releases all the record locks for the files you have under commitment control.
- Repositions the file to its position at the time of the previous COMMIT operation (or at the time of the file OPEN, if there has been no previous COMMIT operation.)

Commitment control starts when the CL command STRCMTCTL is executed. See the chapter on "Commitment Control" in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for more information.

The file changes and the record-lock releases apply to all the files under commitment control in your activation group or job, whether the changes have been requested by the program issuing the ROLBK operation or by another program in the same activation group or job. The program issuing the ROLBK operation does not need to have any files under commitment control. For example, suppose program A calls program B and program C. Program B has files under commitment control, and program C does not. A ROLBK operation in program C still affects the files changed by program B.

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/Errors" on page 96.

For information on how the rollback function is performed by the system, refer to *Recovering your system*, SC41-5304-10.

For more information, see "File Operations" on page 453.

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 string (base string) contained in factor 2 for a substring (compare string) contained in factor 1. The scan begins at a specified location contained in factor 2 and continues for the length of the compare string which is specified in factor 1. The compare string and base string must both be of the same type, either both character, both graphic, or both UCS-2.

Factor 1 must contain either the compare string or the compare string, followed by a colon, followed by the length. The compare string portion of factor 1 can contain one of: a field name, array element, named constant, data structure name, literal, or table name. The length portion must be numeric with no decimal positions and can contain one of: a named constant, array element, field name, literal, or table name. If no length is specified, it is that of the compare string.

Factor 2 must contain either the base string or the base string, followed by a colon, followed by the start location of the SCAN. The base string portion of factor 2 can contain one of: a field name, array element, named constant, data structure name, literal, or table name. The start location portion 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 graphic or UCS-2 strings are used, the start position and length are measured in double bytes. If no start location is specified, a value of 1 is used.

The result field contains the numeric value of the leftmost position of the compare string in the base string, if found. It must be numeric with no decimal positions and can contain one of: a field name, array element, array name, or table name. 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.

Notes:

1. The strings are indexed from position 1.
2. 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.
3. Figurative constants cannot be used in the factor 1, factor 2, or result fields.
4. No overlapping within data structures is allowed for factor 1 and the result field or factor 2 and the result field.

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/Errors” on page 96.

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.

The SCAN begins at the leftmost character of 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. If the result field is not an array, the SCAN operation will locate only the first occurrence of the compare string. To continue scanning beyond the first occurrence, use the result field from the previous SCAN operation to calculate the starting position of the next SCAN. 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; if the result field is an array, all its elements are set to zero.

Leading, trailing, or embedded blanks specified in the compare string are included in the SCAN operation.

The SCAN operation is case-sensitive. A compare string specified in lowercase will not be found in a base string specified in uppercase.

For more information, see “String Operations” on page 468.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The SCAN operation finds the substring 'ABC' starting in
 * position 3 in factor 2;  3 is placed in the result field.
 * Indicator 90 is set on because the string is found.  Because
 * no starting position is specified, the default of 1 is used.
C     'ABC'         SCAN      'XCABCD'      RESULT                  90
 *
 * This SCAN operation scans the string in factor 2 for an
 * occurrence of the string in factor 1 starting at position 3.
 * The 'Y' in position 1 of the base string is ignored because
 * the scan operation starts from position 3.
 * The operation places the values 5 and 6 in the first and
 * 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
 *
 * This SCAN operation scans the string in factor 2, starting
 * at position 2, for an occurrence of the string in factor 1
 * for a length of 4.  Because 'TOOL' is not found in FIELD1,
 * 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
C
 *
 * The SCAN operation is searching for a name.  When the name
 * is found, %FOUND returns '1' so HandleLine is called.
C     SrchName      SCAN      Line
C                   IF        %FOUND
C                   EXSR      HandleLine
C                   ENDIF
```

Figure 374. SCAN Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++
 *
 *       A Graphic SCAN example
 *
 *       Value of Graffld is graphic 'AACCBBGG'.
 *       Value of Number after the scan is 3 as the 3rd graphic
 *       character matches the value in factor 1
D Graffld         S             4G   inz(G'oAACCBBGGi')
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 * The SCAN operation scans the graphic string in factor 2 for
 * an occurrence of the graphic literal in factor 1. As this is a
 * graphic operation, the SCAN will operate on 2 bytes at a time
C
C     G'oBBi'       SCAN      Graffld:2     Number           5 0    90
C
```

Figure 375. 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 group
- 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 specified anywhere in calculations. It 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 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * In the following example, if X equals 1, do the operations in
 * sequence 1 (note that no END operation is needed before the
 * next WHENxx); if X does NOT equal 1, and if Y=2 and X<10, do the
 * operations in sequence 2.  If neither condition is true, do
 * the operations in sequence 3.
 *
C                   SELECT
C                   WHEN      X = 1
C                   Z-ADD     A             B
C                   MOVE      C             D
 *  Sequence 1
C                   :
C                   WHEN      ((Y = 2) AND (X < 10))
 *  Sequence 2
C                   :
C                   OTHER
 *  Sequence 3
C                   :
C                   ENDSL
 *
 * The following example shows a select group with conditioning
 * indicators.  After the CHAIN operation, if indicator 10 is on,
 * then control passes to the ADD operation.  If indicator 10 is
 * off, then the select group is processed.
 *
C     KEY           CHAIN     FILE                              10
C  N10              SELECT
C                   WHEN      X = 1
 *  Sequence 1
C                   :
C                   WHEN      Y = 2
 *  Sequence 2
C                   :
C                   ENDSL
C                   ADD       1              N
```

Figure 376. SELECT 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 key or relative record number specified in factor 1. The file must be a full procedural file (identified by an F in position 18 of the file description specifications).

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 289 on page 635 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 546 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.

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 79.

If the SETGT operation is not successful (no-record-found condition), the file is positioned to the end of the file.

Figurative constants can also be used to position the file.

Note: The discussion and examples of using figurative constants which follow, assume that *LOVAL and *HIVAL are not used as actual keys in the file.

When used with a file with a composite key, figurative constants are treated as though each field of the key contained the figurative constant value. In most cases, *LOVAL positions the file so that the first read retrieves the record with the lowest key. In most cases, *HIVAL positions the file so that a READ receives an end-of-file indication; a subsequent READP retrieves the last record in the file. However, note the following cases for using *LOVAL and *HIVAL:

- With an externally described file that has a key in descending order, *HIVAL positions the file so that the first read operation retrieves the first record in the file (the record with the highest key), and *LOVAL positions the file so that a READP operation retrieves the last record in the file (the record with the lowest key).

- If a record is added or a key field is altered after a SETGT operation with either *LOVAL or *HIVAL, the file may no longer be positioned to the record with the lowest or highest key. key value X'99...9D' and *HIVAL for numeric keys represents a key value X'99...9F'. If the keys are float numeric, *LOVAL and *HIVAL are defined differently. See "Figurative Constants" on page 134. When a program described file has a packed decimal key specified in the file specifications but the actual file key field contains character data, records may have keys that are less than *LOVAL or greater than *HIVAL. When a key field contains unsigned binary data, *LOVAL may not be the lowest key.

When *LOVAL or *HIVAL are used with key fields with a Date or Time data type, the values are dependent of the Date-Time format used. For details on these values please see Chapter 9, "Data Types and Data Formats," on page 179.

Following the SETGT operation, a 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*). You retrieve this record by reading the file. Before you read the file, however, records may be deleted from the file by another job or through another file in your job. Thus, you may not get the record you expected. For information on preventing unexpected modification of your files, see the discussion of allocating objects in the iSeries Information Center Programming topic at URL http://www.ibm.com/systems/i/infocenter/.

See "Database Null Value Support" on page 219 for information on handling records with null-capable fields and keys.

For more information, see "File Operations" on page 453.

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...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *  This example shows how to position the file so READ will read
 *  the next record. The search argument, KEY, specified for the
 *  SETGT operation has a value of 98; therefore, SETGT positions
 *  the file before the first record of file format FILEA that
 *  has a key field value greater than 98.  The file is positioned
 *  before the first record with a key value of 100.  The READ
 *  operation reads the record that has a value of 100 in its key
 *  field.
C
C     KEY           SETGT     FILEA
C                   READ      FILEA                              64
 *
 *  This example shows how to read the last record of a group of
 *  records with the same key value and format from a program
 *  described file.  The search argument, KEY, specified for the
 *  SETGT operation positions the file before the first record of
 *  file FILEB that has a key field value greater than 70.
 *  The file is positioned before the first record with a key
 *  value  of 80.  The READP operation reads the last record that
 *  has a value of 70 in its key field.
C
C     KEY           SETGT     FILEB
C                   READP     FILEB                              64
```

Figure 377. SETGT Operation (Part 1 of 4)

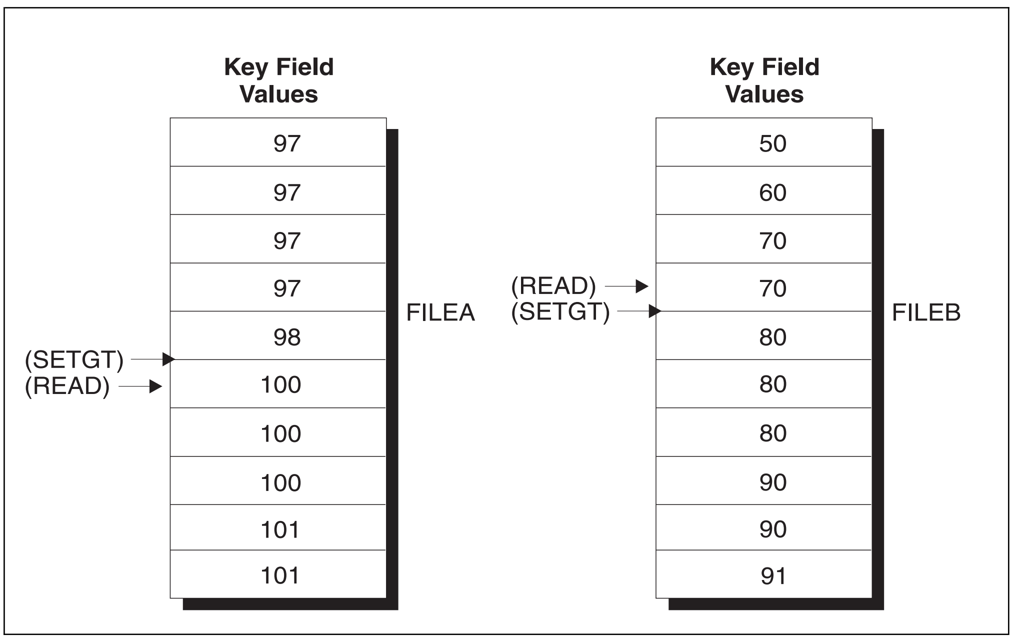


Figure 377. SETGT Operation (Part 2 of 4)

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  This example shows the use of *LOVAL.  The SETLL operation
 *  positions the file before the first record of a file in
 *  ascending order.  The READ operation reads the first record
 *  (key value 97).
C
C     *LOVAL        SETLL     RECDA
C                   READ      RECDA                                  64
C
 *  This example shows the use of *HIVAL.  The SETGT operation
 *  positions the file after the last record of a file in ascending
 *  order. The READP operation reads the last record (key value 91).
C
C     *HIVAL        SETGT     RECDB
C                   READP     RECDB                                  64
```

Figure 377. SETGT Operation (Part 3 of 4)

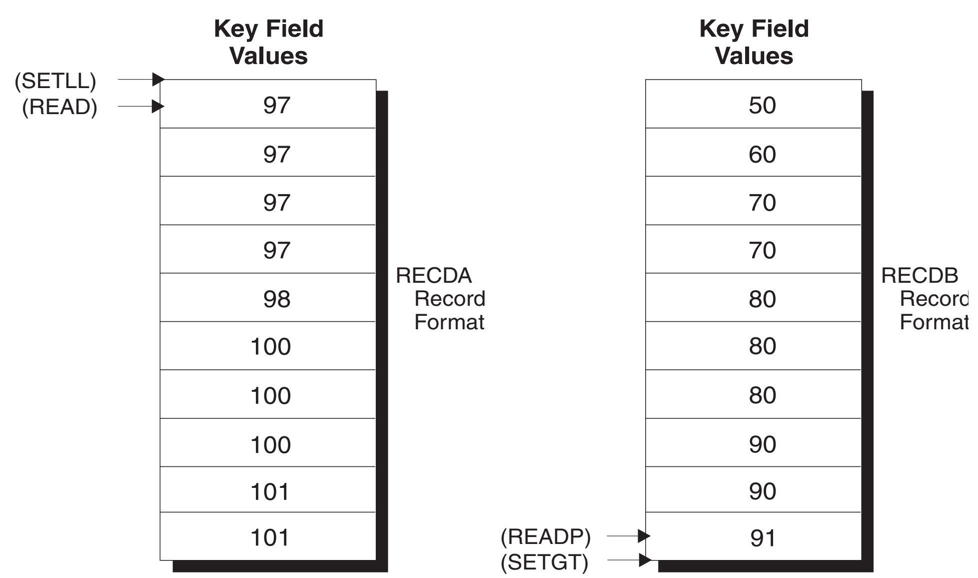


Figure 377. SETGT Operation (Part 4 of 4)

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 (key or relative record number) operand specified (*search-arg*). The file must be a full procedural file (identified by an F in position 18 of the file description specifications).

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 289 on page 635 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 546 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.

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 79.

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.

When using SETLL with an indicator in positions 75 and 76 or with %EQUAL, the comparison between the specified key and the actual key in the file is normally done by data management. In some cases this is impossible, causing the comparison to be done using the hexadecimal collating sequence. This can give different results than expected. For more information, see the section "Unexpected Results Using Keyed Files" in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

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 specified (*search-arg*).

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 specified (*search-arg*).

Figurative constants can be used to position the file.

Note: The discussion and examples of using figurative constants which follow, assume that *LOVAL and *HIVAL are not used as actual keys in the file.

When used with a file with a composite key, figurative constants are treated as though each field of the key contained the figurative constant value. Using SETLL with *LOVAL positions the file so that the first read retrieves the record with the lowest key. In most cases (when duplicate keys are not allowed), *HIVAL positions the file so that a READP retrieves the last record in the file, or a READ receives an end-of-file indication. However, note the following cases for using *LOVAL and *HIVAL:

- With an externally described file that has a key in descending order, *HIVAL positions the file so that the first read operation retrieves the first record in the file (the record with the highest key), and *LOVAL positions the file so that a READP operation retrieves the last record in the file (the record with the lowest key).
- If a record is added or a key field altered after a SETLL operation with either *LOVAL or *HIVAL, the file may no longer be positioned to the record with the lowest or highest key.
- *LOVAL for numeric keys represents a key value X'99...9D' and *HIVAL represents a key value X'99...9F'. If the keys are float numeric, *HIVAL and *LOVAL are defined differently. See "Figurative Constants" on page 134. When a program described file has a packed decimal key specified in the file specifications but the actual file key field contains character data, records may have keys that are less than *LOVAL or greater than *HIVAL. When a key field contains unsigned binary data, *LOVAL may not be the lowest key.

When *LOVAL or *HIVAL are used with key fields with a Date or Time data type, the values are dependent of the Date-Time format used. For details on these values please see Chapter 9, "Data Types and Data Formats," on page 179.

You can use the special values *START and *END for *search-arg*. *START positions to the beginning of the file and *END positions to the end of the file. Both positionings are independent of the collating sequence used for keyed files. If you specify either *START or *END for *search-arg*, note the following:

- | The name of the file must be specified as the *name* operand.
- Either an error indicator (positions 73-74) or the 'E' extender may be specified.

Figure 377 on page 806 (part 3 of 4)shows the use of figurative constants with the SETGT operation. Figurative constants are used the same way with the SETLL operation.

Remember the following when using the SETLL operation:

- If the SETLL operation is not successful (no records found condition), the file is positioned to the end of the file.
- 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, you retrieve this record by reading the file. Before you read the file, however, records may be

deleted from the file by another job or through another file in your job. Thus, you may not get the record you expected. 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 you found a matching record, you may not get that record. For information on preventing unexpected modification of your files, see the discussion of allocating objects in the iSeries Information Center Programming topic at URL http://www.ibm.com/systems/i/infocenter/..

- SETLL does not cause the system to access a data record. If you are only interested in verifying that a key actually exists, SETLL with an equal indicator (positions 75-76) or the %EQUAL built-in function is a better performing solution than the CHAIN operation in most cases. Under special cases of a multiple format logical file with sparse keys, CHAIN can be a faster solution than SETLL.

See “Database Null Value Support” on page 219 for information on handling records with null-capable fields and keys.

For more information, see “File Operations” on page 453.

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...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  All the 101 records in ORDFIL are to be printed.  The value 101
 *  has previously been placed in ORDER.  The SETLL operation
 *  positions the file at the first record with the key value 101
 *  and %EQUAL will return '1'.
C
C     ORDER         SETLL     ORDFIL
C
 *  The following DO loop processes all the records that have the
 *  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
 *  The READE operation reads the second, third, and fourth 101
 *  records in the same manner as the first 101 record was read.
 *  After the fourth 101 record is read, the READE operation is
 *  attempted.  Because the 102 record is not of the same group,
 *  %EOF will return '1', the EXCEPT operation is bypassed, and
 *  the DOU loop ends.
```

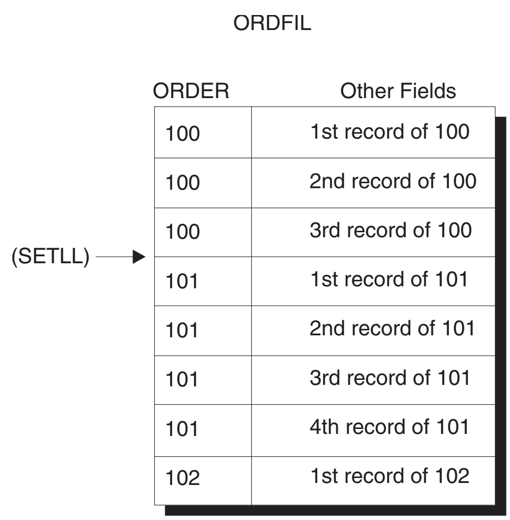


Figure 378. SETLL Operation

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. Entries of 1P and MR are not valid. Setting off L1 through L9 indicators does not automatically set off any lower control-level indicators.

Figure 379 on page 813 illustrates the SETOFF operation.

For more information, see “Indicator-Setting Operations” on page 456.

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. Entries of 1P, MR, KA through KN, and KP through KY are not valid. Setting on L1 through L9 indicators does not automatically set on any lower control-level indicators.

For more information, see "Indicator-Setting Operations" on page 456.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The SETON and SETOFF operations set from one to three indicators
 *  specified in positions 71 through 76 on and off.
 *  The SETON operation sets indicator 17 on.
C
C                   SETON                                        17
C
 *  The SETON operation sets indicators 17 and 18 on.
C
C                   SETON                                        1718
C
 *  The SETOFF operation sets indicator 21 off.
C
C                   SETOFF                                       21
```

Figure 379. SETON and SETOFF Operations

SHTDN (Shut Down)

Free-Form Syntax	(not allowed - use the %SHUT built-in function)

Code	Factor 1	Factor 2	Result Field	Indicators		
SHTDN				ON	_	_

The SHTDN operation allows the programmer to determine whether the system operator has requested shutdown. If the system operator has requested shutdown, the resulting indicator specified in positions 71 and 72 is set on. Positions 71 and 72 must contain one of the following indicators: 01 through 99, L1 through L9, U1 through U8, H1 through H9, LR, or RT.

The system operator can request shutdown by specifying the *CNTRLD option on the following CL commands: ENDJOB (End Job), PWRDWNSYS (Power Down System), ENDSYS (End System), and ENDSBS (End Subsystem). For information on these commands, see the iSeries Information Center programming category.

Positions 73 through 76 must be blank.

For more information, see "Information Operations" on page 457.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  When the SHTDN operation is run, a test is made to determine
 *  whether the system operator has requested shutdown.  If so,
 *  indicator 27 is set on.
C
C                   SHTDN                                        27
C   27              EXSR      Term_appl
C                   :
C                   :
C     Term_appl     BEGSR
C                   CLOSE     *ALL
C                   :
C                   ENDSR
```

Figure 380. SHTDN Operation

SORTA (Sort an Array)

Free-Form Syntax	SORTA{(A/D)} *array-name* | *keyed-ds-array*
	SORTA{(A/D)} %SUBARR(*array-name* | *keyed-ds-array*: *start-element* { : *number-of-elements* })

Code	Factor 1	Extended Factor 2
SORTA(A/D)		Array or keyed-ds-array
		%SUBARR(Array or keyed-ds-array : start-element {:number-of-elements})

For a scalar array, the *array-name* operand is the name of an array to be sorted. The array *IN cannot be specified. If the array is defined as a compile-time or prerun-time array with data in alternating form, the alternate array is not sorted. Only the array specified as *array-name* is sorted.

For an array data structure, the *keyed-ds-array* operand is a qualified name consisting of the array to be sorted followed by the subfield to be used as a key for the sort. The array data structure to be sorted is indicated by specifying * as the index for the array. For example, if array data structure INFO has subfields NAME and SALARY, then to sort array INFO using subfield NAME as a key, specify INFO(*).NAME as the operand for SORTA. To sort the INFO array by SALARY, specify INFO(*).SALARY as the operand for SORTA.

If the sequence for the array is defined by the ASCEND or DESCEND keyword on the definition specification for the array, then the array is always sorted in that sequence. If no sequence is specified for the array, then the sequence defaults to ascending sequence. If the 'A' operation extender is specified, then the array is sorted in ascending sequence. If the 'D' operation extender is specified, then the array is sorted in descending sequence.

Note: The ASCEND and DESCEND keywords cannot be specified for an array data structure.

If the array is defined with the OVERLAY keyword and the 'A' or 'D' operation extender is not specified, the base array will be sorted in the sequence defined by the OVERLAY array.

Graphic and UCS-2 arrays will be sorted by the hexadecimal values of the array elements, disregarding the alternate collating sequence, in the order specified on the definition specification.

To sort a portion of an array, use the %SUBARR built-in function.

Notes:

1. Sorting an array does not preserve any previous order. For example, if you sort an array twice, using different overlay arrays, the final sequence will be that of the last sort. Elements that are equal in the sort sequence but have different hexadecimal values (for example, due to alternate collating sequence or the use of an overlay array to determine sequence), may not be in the same order after sorting as they were before.

2. 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 445 for more information.
3. If a null-capable array is sorted, the sorting will not take the settings of the null flags into consideration.
4. Sorting a dynamically allocated array without all defined elements allocated may cause errors to occur. Use the %SUBARR built-in function to limit the sort to only the allocated elements.
5. The 'A' operation extender is not allowed when sorting an array that is defined with the DESCEND keyword and the 'D' operation extender is not allowed when sorting an array that is defined with the ASCEND keyword.
6. When sorting an array data structure:
   a. The part of the qualified name preceding the (*) index must represent an array, and the part of the qualified name after the (*) must represent a scalar subfield or an indexed scalar array.
   b. If there is more than one array subfield in a complex qualified name, only one array subfield can be sorted. All other arrays in the qualified name must have an index specified. For example, if array data structure FAMILY has an array subfield CHILD and the CHILD elements have an array subfield PET, and the PET subfield has a subfield NAME, then only one of the FAMILY, CHILD and PET arrays can be sorted in one SORTA operation. If the CHILD array is to be sorted, then the FAMILY and PET arrays must have explicit indexes. One valid operand for SORTA would be FAMILY(i).CHILD(*).PET(1).NAME. That SORTA operation would sort the CHILD array of FAMILY(i) by the NAME subfield of PET(1).
   c. An array data structure is sorted in the ascending sequence of the key unless the 'D' operation extender is specified.
   d. If the sort key is an element of a sequenced array, its sequence is not considered when sorting the array data structure.

For more information, see "Array Operations" on page 438.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
DARRY           S              1A   DIM(8) ASCEND
DARRY2          S              1A   DIM(8)
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The SORTA operation sorts ARRY into ascending sequence because
 *  the ASCEND keyword is specified.
 *  If the unsorted ARRY contents were GT1BA2L0, the sorted ARRY
 *  contents would be ABGLT012.
C                   SORTA     ARRY

 *  The SORTA operation sorts ARRY2 into descending ascending sequence
 *  the (D) operation extender is specified.
 *  If the unsorted ARRY2 contents were GT1BA2L0, the sorted ARRY2
 *  contents would be 210TLGBA.
C                   SORTA(D)  ARRY2
```

Figure 381. SORTA Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++
 * In this example, the base array has the values aa44 bb33 cc22 dd11
 * 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
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C
 * After the SORTA operation, the base array has the values
 * dd11 cc22 bb33 aa44
C
C                   SORTA     ARRO
```

Figure 382. SORTA Operation with OVERLAY

```
 * The names array does not have a sequence keyword
 * (ASCEND or DESCEND) specified.
D info            DS                  QUALIFIED
D   names                       10A   DIM(2)

 /free

    // Initialize the array
    info.names(1) = 'Bart';
    info.names(2) = 'Lisa';

    // Sort the info.names in descending order
    SORTA(D) info.names;
    // info.names(1) = 'Lisa'
    // info.names(2) = 'Bart'

    // Sort the info.names in ascending order
    SORTA(A) info.names;
    // info.names(1) = 'Bart'
    // info.names(2) = 'Lisa'

    // With no operation extender, it defaults to ascending order
    SORTA info.names;
    // info.names(1) = 'Bart'
    // info.names(2) = 'Lisa'
```

Figure 383. SORTA Operation Ascending or Descending

```
D emp             DS                  QUALIFIED DIM(25)
D   name                         25A   VARYING
D   salary                        9P 2
D numEmp          S              10I 0

// Initialize the data structure
emp(1).name = 'Maria';
emp(1).salary = 1100;
emp(2).name = 'Pablo';
emp(2).salary = 1200;
emp(3).name = 'Bill';
emp(3).salary = 1000;
emp(4).name = 'Alex';
emp(4).salary = 1300;
numEmp = 4;

// Sort the EMP array using the NAME subfield as a key
SORTA %subarr(emp(*).name : 1 : numEmp);
// emp(1).name = 'Alex'       <-----
// emp(1).salary = 1300
// emp(2).name = 'Bill'       <-----
// emp(2).salary = 1000
// emp(3).name = 'Maria'      <-----
// emp(3).salary = 1100
// emp(4).name = 'Pablo'      <-----
// emp(4).salary = 1200

// Sort the EMP array using the SALARY subfield as a key
SORTA %subarr(emp(*).salary : 1 : numEmp);
// emp(1).name = 'Bill'
// emp(1).salary = 1000       <-----
// emp(2).name = 'Maria'
// emp(2).salary = 1100       <-----
// emp(3).name = 'Pablo'
// emp(3).salary = 1200       <-----
// emp(4).name = 'Alex'
// emp(4).salary = 1300       <-----

// Sort the EMP array descending using the SALARY subfield
SORTA(D) %subarr(emp(*).salary : 1 : numEmp);
// emp(1).name = 'Alex'
// emp(1).salary = 1300       <-----
// emp(2).name = 'Pablo'
// emp(2).salary = 1200       <-----
// emp(3).name = 'Maria'
// emp(3).salary = 1100       <-----
// emp(4).name = 'Bill'
// emp(4).salary = 1000       <-----
```

Figure 384. SORTA Operation with an Array Data Structure

```
D emp_t           DS                  QUALIFIED TEMPLATE
D   name                       25A   VARYING
D teams           DS                  QUALIFIED DIM(2)
D   manager                    25A   VARYING
D   emps                              LIKEDS(emp_t) DIM(2)

// Initialize the data structure
teams(1).manager = 'Jack';
teams(1).emps(1).name = 'Yvonne';
teams(1).emps(2).name = 'Mary';
teams(2).manager = 'Ann';
teams(2).emps(1).name = 'Wendy';
teams(2).emps(2).name = 'Thomas';

// Sort the TEAMS array using the MANAGER subfield as a key
SORTA teams(*).manager;
//    teams(1).manager = 'Ann'           <-----
//    teams(1).emps(1).name = 'Wendy'
//    teams(1).emps(2).name = 'Thomas'
//    teams(2).manager = 'Jack'          <-----
//    teams(2).emps(1).name = 'Yvonne'
//    teams(2).emps(2).name = 'Mary'

// Sort the TEAMS array using the EMPS(2).NAME subfield as a key
SORTA teams(*).emps(2).name;
//    teams(1).manager = 'Jack'
//    teams(1).emps(1).name = 'Yvonne'
//    teams(1).emps(2).name = 'Mary'     <-----
//    teams(2).manager = 'Ann'
//    teams(2).emps(1).name = 'Wendy'
//    teams(2).emps(2).name = 'Thomas'   <-----

// Sort the TEAMS(1).EMPS array using the NAME subfield as a key
SORTA teams(1).emps(*).name;
//    teams(1).manager = 'Jack'
//    teams(1).emps(1).name = 'Mary'     <-----
//    teams(1).emps(2).name = 'Yvonne'   <-----
//    teams(2).manager = 'Ann'
//    teams(2).emps(1).name = 'Wendy'
//    teams(2).emps(2).name = 'Thomas'

// Sort the TEAMS array first by the MANAGER subfield
// and then by the EMPS.NAME subfields
SORTA teams(*).manager;
for i = 1 to %ELEM(TEAMS);
   SORTA teams(i).emps(*).name;
endfor;
// After the first sort, by MANAGER:
//    teams(1).manager = 'Ann'           <-----
//    teams(1).emps(1).name = 'Wendy'
//    teams(1).emps(2).name = 'Thomas'
//    teams(2).manager = 'Jack'          <-----
//    teams(2).emps(1).name = 'Mary'
//    teams(2).emps(2).name = 'Yvonne'
// After loop with the second sort, by EMPS.NAME:
//    teams(1).manager = 'Ann'
//    teams(1).emps(1).name = 'Thomas'   <----- 1
//    teams(1).emps(2).name = 'Wendy'    <----- 1
//    teams(2).manager = 'Jack'
//    teams(2).emps(1).name = 'Mary'     <----- 2
//    teams(2).emps(2).name = 'Yvonne'   <----- 2
```

Figure 385. SORTA Operation with a Complex Array Data Structure

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 one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric, and can contain one of: 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.

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 RPG IV exception/error handling routine receives control.

For further rules on the SQRT operation, see “Arithmetic Operations” on page 434.

See Figure 172 on page 437 for an example of the SQRT 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, the contents of factor 2 are subtracted from the contents of the result field.

Factor 1 and factor 2 must be numeric, and each can contain one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric, and can contain one of: an array, array element, subfield, or table name.

For rules for the SUB operation, see “Arithmetic Operations” on page 434.

See Figure 172 on page 437 for 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

Subtract a duration

The SUBDUR operation can be used to subtract a duration specified in factor 2 from a field or constant specified in factor 1 and place the 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 then the duration is subtracted from the field specified in the result field.

Factor 2 is required and contains two subfactors. The first is a numeric field, array or constant with zero decimal positions. If the field is negative then the 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 449.

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 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 field is a time field, the result will always 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.

Note: The system places a 15 digit limit on durations. Subtracting a Duration with more than 15 significant digits will cause errors or truncation. These problems can be avoided by limiting the first subfactor in Factor 2 to 15 digits.

Calculate a duration

The SUBDUR operation can also be used to calculate a duration between:

1. Two dates
2. A date and a timestamp
3. Two times
4. A time and a timestamp
5. Two timestamps

The data types in factor 1 and factor 2 must be compatible types as specified above.

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 449.

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/Errors" on page 96.

SUBDUR Examples

```
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 * Determine a LOANDATE which is xx years, yy months, zz days prior to
 * the DUEDATE.
C     DUEDATE       SUBDUR    XX:*YEARS     LOANDATE
C                   SUBDUR    YY:*MONTHS    LOANDATE
C                   SUBDUR    ZZ:*DAYS      LOANDATE
 * Add 30 days to a loan due date
 *
C                   SUBDUR    -30:*D        LOANDUE
 * Calculate the number of days between LOANDATE and DUEDATE.
 * If DUEDATE is after LOANDATE, the value of NUM_DAYS will be positive.
C     DUEDATE       SUBDUR    LOANDATE      NUM_DAYS:*D       5 0
 * Determine the number of months between LOANDATE and DUEDATE.
C     DUEDATE       SUBDUR    LOANDATE      NUM_MONTHS:*M     5 0
```

Figure 386. 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 from factor 2, starting at the location specified in factor 2 for the length specified in factor 1, and places this substring 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 and target strings must both be of the same type, either both character, both graphic, or both UCS-2.

Factor 1 can contain the length value of the string to be extracted from the string specified in factor 2. It must be numeric with no decimal positions and can contain one of: a field name, array element, table name, literal, or named constant.

Factor 2 must contain either the base string, or the base string followed by ':', followed by the start location. The base string portion can contain one of: 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 one of the following: a field name, array element, table name, literal or named constant. If it 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.

The start location and the length of the substring to be extracted must be positive integers. The start location must not be greater than the length of the base string, and the length must not be greater than the length of the base string from the start location. If either or both of these conditions is not satisfied, the operation will not be performed.

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/Errors” on page 96.

The result field must be character, graphic, or UCS-2 and can contain one of the following: 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 will be truncated from the right. If the result field is variable-length, its length does not change.

For more information, see “String Operations” on page 468.

Note: You cannot use figurative constants in the factor 1, factor 2, or result fields. Overlapping is allowed for factor 1 and the result field or factor 2 and the result field. If factor 1 is shorter than the length of the result field, a P specified in the operation extender position indicates that the result field should be padded on the right with blanks after the substring occurs.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The SUBST operation extracts the substring from factor 2 starting
 * at position 3 for a length of 2.  The value 'CD' is placed in the
 * result field TARGET.  Indicator 90 is not set on because no error
 * occurred.
C
C                   Z-ADD     3             T                 2 0
C                   MOVEL     'ABCDEF'      String           10
C     2             SUBST     String:T      Target                 90
 *
 * In this SUBST operation, the length is greater than the length
 * of the string minus the start position plus 1.  As a result,
 * 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
 * In this SUBST operation, 3 characters are substringed starting
 * at the fifth position of the base string.  Because P is not
 * specified, only the first 3 characters of TARGET are
 * 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 387. SUBST Operation (Part 1 of 2)

```
 *
 * This example is the same as the previous one except P
 * specified, and the result is padded with blanks.
 * TARGET equals '123ƀƀƀƀƀ'.
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
 *
 * In the following example, CITY contains the string
 * 'Toronto, Ontario'.  The SCAN operation is used to locate the
 * separating blank, position 9 in this illustration.  SUBST
 * without factor 1 places the string starting at position 10 and
 * continuing for the length of the string in field TCNTRE.
 * TCNTRE contains 'Ontario'.
C     ' '           SCAN      City          C
C                   ADD       1             C
C                   SUBST     City:C        TCntre
 *
 * Before the operations STRING='ƀƀƀJohnƀƀƀ&
 * RESULT is a 10 character field which contains 'ABCDEFGHIJ'.
 * The CHECK operation locates the first nonblank character
 * and sets on indicator 10 if such a character exists.  If *IN10
 * is on, the SUBST operation substrings STRING starting from the
 * first non-blank to the end of STRING.  Padding is used to ensure
 * that nothing is left from the previous contents of the result
 * field.  If STRING contains the value '   HELLO ' then RESULT
 * will contain the value 'HELLO     ' after the SUBST(P) operation.
 * After the operations RESULT='Johnƀƀƀƀƀƀ'.
C
C     ' '           CHECK     STRING        ST                      10
C   10              SUBST(P)  STRING:ST     RESULT
```

Figure 387. SUBST Operation (Part 2 of 2)

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 (Go To)" on page 696 or "CABxx (Compare and Branch)" on page 619 operation. It can be specified anywhere within calculations, including within total calculations.

A GOTO within a subroutine in the cycle-main procedure can be issued to a TAG within the same subroutine, detail calculations or total calculations. 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.

The control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, the LR indicator, or the L0 entry to group the statement within the appropriate section of the program. Conditioning indicator entries (positions 9 through 11) are not allowed.

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.

Branching to the TAG from a different part of the RPG IV logic cycle may result in an endless loop. For example, if a detail calculation line specifies a GOTO operation to a total calculation TAG operation, an endless loop may occur.

See Figure 324 on page 697 for examples of the TAG operation.

For more information, see "Branching Operations" on page 439 or "Declarative Operations" on page 452.

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 code allows users to test the validity of date, time, or timestamp fields prior to using them.

For information on the formats that can be used see "Date Data Type" on page 206, "Time Data Type" on page 208, and "Timestamp Data Type" on page 210.

- If the *field-name* operand is a field declared as Date, Time, or Timestamp:
  - The *dtz-format* operand cannot be specified
  - Operation code extenders 'D', 'T', and 'Z' are not allowed
- If the *field-name* operand is a field declared as character or numeric, then one of the operation code extenders 'D', 'T', or 'Z' must be specified.

  Note: If the *field-name* operand is a character field with no separators, the *dtz-format* operand must contain 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 by any of the valid Date formats (See "Date Data Type" on page 206).
    - 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 be any of the valid Time formats (See "Time Data Type" on page 208).
    - If *dtz-format* is not specified, the format specified on the control specification with the TIMFMT keyword is assumed. 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 be *ISO or *ISO0 (See "Timestamp Data Type" on page 210).

Numeric fields and character fields without separators are tested for valid digit portions 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/Errors” on page 96.

For more information, see “Date Operations” on page 449 or “Test Operations” on page 475.

```
*...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
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  Indicator 18 will not be set on, since the character field is a
 *  valid *ISO timestamp field, without separators.
C     *ISO0         TEST (Z)                Char_Tstmp             18
 *  Indicator 19 will not be set on, since the character field is a
 *  valid *MDY date, without separators.
C     *MDY0         TEST (D)                Char_Date              19
 *
 *  %ERROR will return '1', since Num_Date is not *DMY.
 *
C     *DMY          TEST (DE)               Num_Date
 *
 *  No Factor 1 since result is a D data type field
 *  %ERROR will return '0', since the field
 *  contains a valid date
C
C                   TEST (E)                Datefield
C
 * In the following test, %ERROR will return '1' since the
 * Timefield does not contain a valid USA time.
C
C     *USA          TEST (ET)               Char_Time
C
 *  In the following test, indicator 20 will be set on since the
 *  character field is a valid *CMDY, but there are separators.
C
C     *CMDY0        TEST (D)                char_date2             20
C
 *  In the following test, %ERROR will return '0' since
 *  the character field is a valid *LONGJUL date.
C
C     *LONGJUL      TEST (DE)               char_date3
```

Figure 388. TEST (E D/T/Z) Example

TESTB (Test Bit)

Free-Form Syntax	(not allowed - use the %BITAND built-in function. See Figure 195 on page 502.)

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 389 on page 833 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. That is, 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. That is, 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. That is, 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 439 or "Test Operations" on page 475.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The field bit settings are FieldF = 00000001, and FieldG = 11110001.
 *
 *  Indicator 16 is set on because bit 3 is off (0) in FieldF.
 *  Indicator 17 is set off.
C                   TESTB     '3'          FieldF              16  17
 *
 *  Indicator 16 is set on because both bits 3 and 6 are off (0) in
 *  FieldF.  Indicators 17 and 18 are set off.
C                   TESTB     '36'         FieldF              161718
 *
 *  Indicator 17 is set on because bit 3 is off (0) and bit 7 is on
 *  (1) in FLDF.  Indicators 16 and 18 are set off.
C                   TESTB     '37'         FieldF              161718
 *
 *  Indicator 17 is set on because bit 7 is on (1) in FLDF.
 *  Indicator 16 is set off.
C                   TESTB     '7'           FieldF             16  17
 *
 *  Indicator 17 is set on because bits 0,1,2, and 3 are off (0) and
 *  bit 7 is on (1).  Indicators 16 and 18 are set off.
C                   TESTB     FieldG        FieldF             161718
 *
 *  The hexadecimal literal X'88' (10001000) is used in factor 2.
 *  Indicator 17 is set on because at least one bit (bit 0) is on
 *  Indicators 16 and 18 are set off.
C                   TESTB     X'88'         FieldG             161718
```

Figure 389. 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 low-order character, must contain a hexadecimal F zone and a digit (0 through 9). The low-order character is numeric if it contains a hexadecimal C, hexadecimal D, or hexadecimal F zone, and a digit (0 through 9). Note that the alphabetic characters J through R, should they appear in the low-order position of a field, are treated as negative numbers by TESTN. As a result of the test, resulting indicators are set on as follows:

- *Positions 71 and 72:* The result field contains numeric characters; the low-order character may also be a letter from A to R, since these characters have a zone of C, D, or F, and a digit of 0 to 9.
- *Positions 73 and 74:* The result field contains both numeric characters and at least one leading blank. For example, the values ƀ123 or ƀƀ123 set this indicator on. However, the value ƀ1ƀ23 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 (e.g. arithmetic operations).

For more information, see “Test Operations” on page 475.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The field values are FieldA = 123, FieldB = 1X4, FieldC = 004,
 *  FieldD = ƀƀƀ, FieldE = ƀ1ƀ3, and FieldF = ƀ12.
 *
 *  Indicator 21 is set on because FieldA contains all numeric
 *  characters.
C                   TESTN                   FieldA              21
 *  Indicator 22 is set on because FieldA contains all numeric
 *  characters.  Indicators 23 and 24 remain off.
C                   TESTN                   FieldA              222324
 *  All indicators are off because FieldB does not contain valid
 *  numeric data.
C                   TESTN                   FieldB              252627
 *  Indicator 28 is set on because FieldC contains valid numeric data.
 *  Indicators 29 and 30 remain off.
C                   TESTN                   FieldC              282930
 *  Indicator 33 is set on because FieldD contains all blanks.
 *  Indicators 31 and 32 remain off.
C                   TESTN                   FieldD              313233
 *  Indicators 34, 35, and 36 remain off.  Indicator 35 remains off
 *  off because FieldE contains a blank after a digit.
C                   TESTN                   FieldE              343536
 *  Indicator 38 is set on because FieldF contains leading blanks and
 *  valid numeric characters.  Indicators 37 and 39 remain off.
C                   TESTN                   FieldF              373839
```

Figure 390. TESTN Operation

TESTZ (Test Zone)

Free-Form Syntax	(not allowed - use the %BITAND built-in function with X'F0' to isolate the zone part of the character)

Code	Factor 1	Factor 2	Result Field	Indicators		
TESTZ			Character field	AI	JR	XX

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 76. The characters &, A through I, and any character with the same zone as the character A set on the indicator in positions 71 and 72. The characters - (minus), J through R, and any character with the same zone as the character J set on the indicator in positions 73 and 74. Characters with any other zone set on the indicator in positions 75 and 76.

For more information, see "Test Operations" on page 475.

TIME (Retrieve Time and Date)

Free-Form Syntax	(not allowed – use the %DATE, %TIME, and %TIMESTAMP built-in functions)

Code	Factor 1	Factor 2	Result Field	Indicators		
TIME			Target field			

The TIME operation accesses the system time of day and/or the system date at any time during program processing. The system time is based on the 24-hour clock.

The Result field can specify one of the following into which the time of day or the time of day and the system date are written:

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

If the Result field is a numeric field, 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)

If the Result field is a numeric field, then if the system date is included, it is placed in positions 7 through 12 or 7 through 14 of the result field. 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.

Note: 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.

For more information, see “Information Operations” on page 457.

```
D   Timeres       S             T   TIMFMT(*EUR)
D   Dateres       S             D   DATFMT(*USA)
D   Tstmpres      S             Z
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  When the TIME operation is processed (with a 6-digit numeric
 *  field), the current time (in the form hhmmss) is placed in the
 *  result field CLOCK.  The TIME operation is based on the 24-hour
 *  clock, for example, 132710.  (In the 12-hour time system, 132710
 *  is 1:27:10 p.m.)
C                   TIME                    Clock            6 0
 *  When the TIME operation is processed (with a 12-digit numeric
 *  field), the current time and day is placed in the result field
 *  TIMSTP.  The first 6 digits are the time, and the last 6 digits
 *  are the date; for example, 093315121579 is 9:33:15 a.m. on
 *  December 15, 1979.
C                   TIME                    TimStp          12 0
C                   MOVEL     TimStp        Time             6 0
C                   MOVE      TimStp        SysDat           6 0
 *  This example duplicates the 12-digit example above but uses a
 *  14-digit field.  The first 6 digits are the time, and the last
 *  8 digits are the date;   for example, 13120001101992
 *  is 1:12:00 p.m. on January 10, 1992.
C                   TIME                    TimStp          14 0
C                   MOVEL     TimStp        Time             6 0
C                   MOVE      TimStp        SysDat           8 0
 *  When the TIME operation is processed with a date field,
 *  the current date is placed in the result field DATERES.
 *  It will have the format of the date field.  In this case
 *  it would be in *USA format ie: D'mm/dd/yyyy'.
C                   TIME                    Dateres
 *  When the TIME operation is processed with a time field,
 *  the current time is placed in the result field TIMERES.
 *  It will have the format of the time field.  In this case
 *  it would be in *EUR format ie: T'hh.mm.ss'.
C                   TIME                    Timeres
 *  When the TIME operation is processed with a timestamp field,
 *  the current timestamp is placed in the result field TSTMPRES.
 *  It will be in *ISO format.
 *  ie: Z'yyyy-mm-dd-hh.mm.ss.mmmmmm'
C                   TIME                    Tstmpres
```

Figure 391. 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 is used to unlock data areas and release 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/Errors” on page 96.

Positions 71,72,75 and 76 must be blank.

For further rules for the UNLOCK operation, see “Data-Area Operations” on page 448.

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 a *DTAARA DEFINE statement or with the DTAARA keyword on the definition specification. *name* must not refer to the local data area or the Program Initialization Parameters (PIP) data area. 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 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  TOTAMT, TOTGRS, and TOTNET are defined as data areas in the
 *  system.  The IN operation retrieves all the data areas defined in
 *  the program.  The program processes calculations, and
 *  then unlocks the data areas. The data areas can them be used
 *  by other programs.
 *
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 392. Data area unlock operation

Releasing record locks

The UNLOCK operation also allows the most recently locked record to be unlocked for an update disk file.

name must be the name of the UPDATE disk file.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++
 *
FUPDATA    UF   E             DISK
 *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  Assume that the file UPDATA contains record format vendor.
 *  A record is read from UPDATA. Since the file is an update
 *  file, the record is locked. *IN50 is set somewhere else in
 *  the program to control whether an UPDATE should take place.
 *  otherwise the record is unlocked using the UNLOCK operation.
 *  Note that factor 2 of the UNLOCK operation is the file name,
 *  UPDATA, not the record format, VENDOR
 *
C                   READ      VENDOR                                 12
C                   :
C     *IN50         IFEQ      *ON
C                   UPDATE    VENDOR
C                   ELSE
C                   UNLOCK    UPDATA                               99
C                   ENDIF
```

Figure 393. 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 or record format)	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 the record and the UPDATE operation.

The *name* operand must be the name of a file or record format to be updated. A record format name is required with an externally described file. The record format name must be the name of the last record read from the file; otherwise, an error occurs. A file name as the *name* operand is required with a program described file.

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 database file, the data structure must be a data structure defined from the same file or record format, with *INPUT or *OUTPUT specified as the second parameter of the LIKEREC or EXTNAME keyword.
4. If *name* refers to a subfile record format from an externally described display file, the data structure must be a data structure defined from the same file or record format, with *OUTPUT specified as the second parameter of the LIKEREC or EXTNAME keyword.
5. See “File Operations” on page 453 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 533 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 79.

Remember the following when using the UPDATE operation:

- When *name* is a record format name, the current values in the program for the fields in the record definition are used to modify the record.
- 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.
- Before UPDATE is issued to a file or record, a valid input operation with lock (READ, READC, READE, READP, READPE, CHAIN, or primary/secondary file) must be issued to the same file or record. If the read operation returns with an

error condition or if it was read without locking, 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.

- 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.
- Beware of using the UPDATE operation on primary or secondary files during total calculations. At this stage in the RPG IV cycle, the fields from the current record (the record that is about to be processed) have not yet been moved to the processing area. Therefore, the UPDATE operation updates the current record with the fields from the preceding record. Also, when the fields from the current record are moved to the processing area, they are the fields that were updated from the preceding record.
- For multiple device files, specify a subfile record format as the *name* operand. The operation is processed for the program device identified in the fieldname specified using the DEVID keyword in the file specification. If the program device is not specified, the device used in the last successful input operation is used. This device must be the same one you specified for the input operation that must precede the UPDATE operation. You must not process input or output operations to other devices in between the input and UPDATE operations. If you do, your UPDATE operation will fail.
- For a display file which has multiple subfile record formats, you must not process read-for-update operations to one subfile record in between the input and UPDATE operations to another subfile in the same display file. If you do, the UPDATE operation will fail.
- An UPDATE operation is valid to a subfile record format as long as at least one successful input operation (READC, CHAIN) has occurred to that format name without an intervening input operation to a different format name. The record updated will be the record retrieved on the last successful input operation. This means that if you read a record successfully, then read unsuccessfully to the same format, an update will succeed, but will update the first record. This is different from the behavior of DISK files.

  To avoid updating the wrong record, check the resulting indicator or record-identifying indicator to ensure that a successful input operation has occurred before doing the update operation.

See “Database Null Value Support” on page 219 for information on updating records with null-capable fields containing null values.

For more information, see “File Operations” on page 453.

WHEN (When True Then Select)

Free-Form Syntax	WHEN{(MR)} *indicator-expression*

Code	Factor 1	Extended Factor 2
WHEN (M/R)		indicator-expression

The WHEN 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 expression in the *indicator-expression* operand is true. See Chapter 20, "Expressions," on page 477 for details on expressions. For information on how operation extenders M and R are used, see "Precision Rules for Numeric Operations" on page 486.

For more information, see "Compare Operations" on page 445 or "Structured Programming Operations" on page 469.

```
CL0N01Factor1+++++++Opcode(E)+Extended-factor2+++++++++++++++++++++++++++..
 *
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 394. 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 (Begin a Select Group)” on page 802 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.
- Within total calculations, the control level entry (positions 7 and 8) can be blank or can contain an L1 through L9 indicator, an LR indicator, or an L0 entry to group the statement within the appropriate section of the program. The control level entry is for documentation purposes only. Conditioning indicator entries (positions 9 through 11) are not allowed.

Refer to “Compare Operations” on page 445 for valid values for xx.

For more information, see “Structured Programming Operations” on page 469.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following example shows nested SELECT groups.  The employee
 * type can be one of 'C' for casual, 'T' for retired, 'R' for
 * regular, and 'S' for student.  Depending on the employee type
 * (EmpTyp), the number of days off per year (Days) will vary.
 *
C                   SELECT
C     EmpTyp        WHENEQ    'C'
C     EmpTyp        OREQ      'T'
C                   Z-ADD     0             Days
C     EmpTyp        WHENEQ    'R'
 *
 * When the employee type is 'R', the days off depend also on the
 * number of years of employment.  The base number of days is 14.
 * For less than 2 years, no extra days are added.  Between 2 and
 * 5 years, 5 extra days are added.  Between 6 and 10 years, 10
 * extra days are added, and over 10 years, 20 extra days are added.
 *
C                   Z-ADD     14            Days
 * 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
 * End of nested select group.
C     EmpTyp        WHENEQ    'S'
C                   Z-ADD     5             Days
C                   ENDSL
```

Figure 395. WHENxx Operation (Part 1 of 2)

```
 *-------------------------------------------------------------------
 * Example of a SELECT group with complex WHENxx expressions.  Assume
 * that a record and an action code have been entered by a user.
 * Select one of the following:
 *   - When F3 has been pressed, do subroutine QUIT.
 *   - When action code(Acode) A (add) was entered and the record
 *     does not exist (*IN50=1), write the record.
 *   - When action code A is entered, the record exists, and the
 *     active record code for the record is D (deleted); update
 *     the record with active rec code=A.  When action code D is
 *     entered, the record exists, and the action code in the
 *     record (AcRec) code is A; mark the record as deleted.
 *   - When action code is C (change), the record exists, and the
 *     action code in the record (AcRec) code is A; update the record.
 *   - Otherwise, do error processing.
 *-------------------------------------------------------------------
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
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 395. WHENxx Operation (Part 2 of 2)

WRITE (Create New Records)

Free-Form Syntax	WRITE{(E)} *name* {*data-structure*}

Code	Factor 1	Factor 2	Result Field	Indicators		
WRITE (E)		name (file or record format)	data-structure	_	ER	EOF

The WRITE operation writes a new record to a file.

The name operand must be the name of a program-described file or a record format from an externally-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 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 453 for information on how to define the data structure and how data is transferred between the file and the data structure.

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 79.

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.

When using the WRITE operation remember:
- When *name* is a record format name, the current values in the program for all the fields in the record definition are used to construct the record.
- When records that use relative record numbers are written to a file, you must update the field name specified on the RECNO File specification keyword (relative record number), so it contains the relative record number of the record to be written.
- When you use the WRITE operation to add records to a DISK file, you must specify an A in position 20 of the file description specifications. (See "Position 20 (File Addition)" on page 283.)
- Device dependent functions are limited. For example, if a "WRITE" is issued to a "PRINTER" device, the space after will be set to 1 if the keyword PRTCTL is not specified on the file specification (normally spacing or skipping information are specified in columns 41 through 51 of the output specifications). If the file is externally described, these functions are part of the external description.
- For a multiple device file, data is written to the program device named in the field name specified with the DEVID keyword on the file description

specifications. (See "DEVID(fieldname)" on page 293.) If the DEVID keyword is not specified, data is written to the program device for which the last successful input operation was processed.

See "Database Null Value Support" on page 219 for information on adding records with null-capable fields containing null values.

For more information, see "File Operations" on page 453.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 *  The WRITE operation writes the fields in the data structure
 *  DS1 to the file, FILE1.
 *
C                   WRITE     FILE1         DS1
```

Figure 396. WRITE Operation

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 434 or "Array Operations" on page 438.

See Figure 172 on page 437 for 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	_

Characters in the source string (factor 2) are translated according to the From and To strings (both in factor 1) and put into a receiver field (result field). Source characters with a match in the From string are translated to corresponding characters in the To string. The From, To, Source, and Target strings must be of the same type, either all character, all graphic, or all UCS-2. As well, their CCSIDs must be the same, unless one of the CCSIDs is 65535, or in the case of graphic fields, CCSID(*GRAPH : *IGNORE) was specified on the Control Specification.

XLATE starts translating the source at the location specified in factor 2 and continues character by character, from left to right. If a character of 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.

Factor 1 must contain the From string, followed by a colon, followed by the To string. The From and To strings can contain one of the following: a field name, array element, named constant, data structure name, literal, or table name.

Factor 2 must contain either the source string or the source string followed by a colon and the start location. The source string portion of factor 2 can contain one of the following: 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 location portion 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 location is specified, a value of 1 is used.

The result field can be a field, array element, data structure, or 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 will be left adjusted. If the result field is shorter than the source string, the result field will contain the leftmost part of the translated source. If the result field is variable-length, its length does not change.

If a character in the From string is duplicated, the first occurrence (leftmost) is used.

Note: 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.

If the From string is longer than the To string, the additional characters in the From string are ignored.

Any valid indicator can be specified in columns 7 to 11.

If factor 2 is shorter than the result field, a P specified in the operation extender position indicates that the result field should be padded on the right with blanks after the translation. If the result field is graphic and P is specified, graphic blanks will 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/Errors” on page 96.

Columns 75-76 must be blank.

For more information, see “String Operations” on page 468.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
 *
 * The following translates the blank in NUMBER to '-'.  The result
 * in RESULT will be '999-9999'.
 *
C                   MOVE      '999 9999'    Number            8
C     ' ':'-'       XLATE     Number        Result            8
```

Figure 397. XLATE Operation

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...+....
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
D Up              C                   'ABCDEFGHIJKLMNOPQRS-
D                                     'TUVWXYZ'
D Lo              C                   'abcdefghijklmnopqrs-
                                      'tuvwxyz'
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..
 *
 * In the following example, all values in STRING are translated to
 * uppercase.  As a result, RESULT='RPG DEPT'.
 *
C                   MOVE      'rpg dept'    String            8
C     Lo:Up         XLATE     String        Result
 *
 * In the following example only part of the string is translated
 * to lowercase.  As a result, RESULT='RPG Dept'.
 *
C     Up:Lo         XLATE     String:6      Result
```

Figure 398. XLATE Operation With Named Constants

XML-INTO (Parse an XML Document into a Variable)

Free-Form Syntax	XML-INTO{(EH)} *receiver* %XML(*xmlDoc* {: *options* });
	XML-INTO{(EH)} %HANDLER(*handlerProc* : *commArea*) %XML(*xmlDoc* {: *options* });

Code	Factor 1	Extended Factor 2
XML-INTO		*receiver* %XML(*xmlDoc* {: *options* })
XML-INTO		%HANDLER(*handlerProc* : *commArea*) %XML(*xmlDoc* {: *options* })

Tip: If you are not familiar with the basic concepts of XML and of processing XML documents, you may find it helpful to read the "Processing XML Documents" section in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* before reading further in this section.

XML-INTO can operate in two different ways:
- Reading XML data directly into an RPG variable
- Reading XML data gradually into an array parameter that it passes to the procedure specified by %HANDLER(handlerProc).

The first operand specifies the target of the parsed data. It can contain a variable name or the %HANDLER built-in function.

The second operand must be the %XML built-in function, identifying the XML document to be parsed and the options controlling the way the parsing is done. See "%XML (xmlDocument {:options})" on page 604 for more information on %XML.

If the first operand is a variable name:
- Parsing will be done directly into the variable.
- The name of the variable will be used to establish the name of the XML element to parse; this can be overridden using the "path option" on page 858.
- If the variable is a data structure, some subfields may be set by the operation even if the operation ends in error.
- If the variable is an array, the parsing will only search for as much data as will fit in the array. The "Number of XML Elements" subfield in positions 372 - 379 of the PSDS will be set to the number of elements successfully set by the operation. For an array of data structures, this value will not include the element being set if a parsing error occurs while parsing the data for the subfields of the element; however, this array element may have some of its subfields set by the operation.

If the first operand is the %HANDLER built-in function:
- The procedure specified as the first operand of %HANDLER will be called when the parser has parsed enough XML data to fill the specified number of RPG array elements handled by the procedure. When the handler returns, the parser will continue to parse the XML data until it has parsed enough XML data to again fill the specified number of array elements to call the handling procedure. This continues until the document is completely parsed, or until the procedure returns a return code indicating that the parsing should halt.

  The final call to the handling procedure may have fewer RPG array elements than the handling procedure can handle. The handling procedure should always refer to the "number of elements" parameter to ensure it does not access array elements that do not have any XML data.

The communication-area variable specified as the second operand of %HANDLER will be passed by the parser as the first parameter to the handling procedure, allowing the procedure coding the XML-INTO operation to communicate with the handling procedure, and allowing the handling procedure to save information from one call to the next.

- Each element of the temporary variable used to hold the array parameter for the procedure will be cleared to its default value before it is loaded from the XML data.
- The *path* option must be used to specify the name of the XML element to search for. See "%XML options for the XML-INTO operation code" on page 856 and "Expected format of XML data" on page 877 and for information about the *path* option.
- The array-handling procedure may be called several times during the XML-INTO operaton. When the parser has found the number of elements specified by the DIM keyword on the second parameter, the procedure will be called. The final time the procedure is called may have fewer elements than specified by the DIM keyword. If there are no elements found, the procedure will not be called.

  The handling procedure must have the following parameters and return type:

Parameter number or return value	Data type and passing mode	Description
Return value	4-byte integer (10I 0)	Returning a value of zero indicates that parsing should continue; returning any other value indictes that parsing should end.
1	Any type, passed by reference	Used to communicate between the XML-INTO operation and the handler, and between successive calls to the handler.
2	Array, or array of data structures, passed by read-only reference (CONST keyword)	The array elements contain the data from the XML elements specified by the *path* option.
3	4-byte unsigned (10U 0), passed by value	The number of array elements in the second parameter that represent XML data.

- See "%HANDLER (handlingProcedure : communicationArea)" on page 539 for more information on %HANDLER.

Subfields of a data structure will be set in the order they appear in the XML document; the order could be important if subfields overlap within the data structure.

%NULLIND is not updated for any field or subfield during an XML-INTO operation.

Operation extender H can be specified to cause numeric data to be assigned half-adjusted. Operation extender E can be specified to handle the following status codes:

00351 Error in XML parsing

00352 Invalid XML option

00353 XML document does not match RPG variable

00354 Error preparing for XML parsing

Note: Operation extenders can be specified only when Free-form syntax is used.

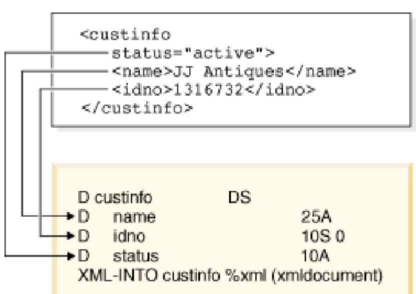
For status 00351, the return code from the parser will be placed in the subfield "External return code" in positions 368-371 of the PSDS. This subfield will be set to zero at the beginning of the operation and set to the value returned by the parser at the end of the operation.

If an unknown, invalid or unrelated option is found in the options parameter of the %XML built-in function, the operation will fail with status code 00352 (Error in XML options). The External return code subfield in the PSDS will not be updated from the initial value of zero, set when the operation begins.

The XML document is expected to match the RPG variable with respect to the names of the XML elements or attributes.

- The XML data for an RPG data structure is expected to have an XML element with the same name as the data structure and child elements and/or attributes with the same names as the RPG subfields.
- The XML data for an RPG array is expected to have a series of elements with the same name as the RPG array.

The *path* option can be used to set the name of the XML element matching the name of the specified variable, but it cannot be used to set the names of the XML elements and/or attributes matching a specified variable's subfields. For example, if variable DS1 has a subfield SF1, the XML element for DS1 can have any name, but the XML element or attribute for SF1 must have the name "sf1" (or "SF1", "Sf1", etc., depending on the *case* option).



When the RPG variable is an array or array of data structures, or when the %HANDLER built-in function is specified, the XML elements corresponding to the array elements are expected to be contained in another XML element. By default, the XML elements will be expected to be child elements of the outermost XML element in the document. The *path* option can be used to specify the exact path to the XML elements corresponding to the array elements. For example, if the outermost XML element is named "transaction", and it has a child element named "parts" which itself has several child elements named "part", then to read the "part" XML elements into an array, you would specify the option 'path=transaction/ parts/part'.

```
<transaction>
  <parts>
     <part type = "bracket" size="15" num="100"/>
     <part type="frame" size="2" num="500"/>
  </parts>
<transaction>
```

When the XML document does not match the RPG variable, for example if the XML document does not contain the default or specified path, or if it is missing some XML elements or attributes to match the subfields of an RPG data structure, the XML-INTO operation will fail with status 00353. The *allowextra* and *allowmissing* options can be used to specify whether an XML element can have more or less data than is required to fully set the RPG variable.

For some RPG data types, XML attributes can be specified to control how the XML data is assigned to the RPG variable. See “Rules for transferring XML data to RPG variables” on page 881 for more information on these attributes.

If an XML reference other than the predefined references &amp, &apos, &lt, &gt, &quot, or the hexadecimal unicode references &#xxxx is found, the result will contain the reference itself, in the form "&refname;". If this value is not valid for the data type, the operation will fail. For example, if an XML element has the value <data>1&decpoint;50/data> the string "1&decpoint;50" would be built up from the three pieces "1", "&decpoint;", and "0". This data is valid for a character or UCS-2 variable, but it would cause an error if converted to other types.

Tip: If XML data is known to contain such references, then following the completion of the XML-INTO operation, character and UCS-2 data should be inspected for the presence of references, and the correct value for the reference substituted using string operations such as %SCANRPL, or %SCAN and %REPLACE.

If XML data is not valid for the type of the RPG variable it matches, the operation will fail with status 0353; the specific status code for the assignment error will appear in the replacement text for message RNX0353.

Tip: To avoid the XML-INTO operation failing because the data cannot be successfully assigned to RPG fields with types such as Date or Numeric, the receiver variable can be defined with subfields that are all of type character or UCS-2. Then the data can be converted to other data types by the RPG program using the conversion built-in functions %DATE, %INT, and so on.

The XML-INTO operation ignores the DOCTYPE declaration. The DOCTYPE declaration may contain the values of entity references that your program will have to handle manually. If you want to have access to the DOCTYPE declaration of the XML document, you can use the XML-SAX operation. Your XML-SAX handling procedure can halt the parsing as soon as it has found the DOCTYPE declaration value, or as soon as it knows that there will be no DOCTYPE declaration.

The following links provide more information on XML-INTO.

- “%XML options for the XML-INTO operation code” on page 856
- “Expected format of XML data” on page 877

- "Rules for transferring XML data to RPG variables" on page 881
- "Examples of the XML-INTO operation" on page 882

%XML options for the XML-INTO operation code

Several options are available for customizing the XML-INTO operation. The options are specified as the second parameter of the %XML built-in function. The parameter can be a constant or a variable expression. The options are specified in the form `'opt1=val1 opt2=val2'`.

See "%XML (xmlDocument {:options})" on page 604 for more information on how to specify the options.

- The *path* option specifies where to locate the desired XML element within the XML document.
- The *doc* option specifies whether the first parameter of the %XML built-in function has an XML document, or has the name of a file containing the XML document.
- The *ccsid* option specifies the CCSID to be used to parse the XML document.
- The *case* option specifies whether the names within the XML document are in lower case, upper case, or mixed case.
- The *trim* option specifies whether you want blanks, tabs and line-end characters to be trimmed from the XML data before it is assigned to your RPG variables.
- The *allow missing* option specifies how the RPG runtime should handle the situation when the XML document does not have enough XML elements or XML attributes to provide data for all the RPG subfields of a data structure.
- The *allow extra* option specifies how the RPG runtime should handle the situation when the XML document has additional XML elements or attributes that are not needed to set the RPG variable.
- The *data subfield* option specifies the name of the extra subfield used to handle the situation where there is text data for an XML element that matches an RPG data structure.
- The *count prefix* option specifies the prefix for the names of the additional subfields that receive the number of RPG array elements set by the XML-INTO operation.

doc (default *string*)

The *doc* option indicates what the source operand of %XML contains.

- *string* indicates that the source operand contains XML data
- *file* indicates that the source operand contains the name of a file in the Integrated File System

```
// In the following example, the first parameter
// of %XML is the name of a file.  Option
// "doc=file" must be specified.
ifsfile = 'myfile.xml';
opt = 'doc=file';
XML-INTO myfield %XML(ifsfile : opt);

// In the following example, the first parameter
// of %XML is an XML document.  Since the "doc"
// option defaults to "string", no options are
// necessary.
xmldata = '<data><num>3</num></data>';
XML-INTO data %XML(xmldata);

// However, "doc=string" may still be specified.
xmldata = '<data><num>3</num></data>';
XML-INTO data %XML(xmldata : 'doc=string');
```

Figure 399. Example of the doc option:

ccsid (default *best*)

The *ccsid* option specifies the CCSID to be used for processing the XML document. Some CCSID conversions may be performed during the XML-INTO operation:

- CCSID conversion may be required from the XML document to a temporary copy of the XML document, if the CCSID of the XML document differs from the CCSID used for parsing.
- CCSID conversion may be required when assigning XML data to an RPG variable, if the CCSID used for parsing differs from the CCSID of the RPG variable.

If the CCSID of the actual document is different from the CCSID to be used for processing the document, CCSID conversion will be done on the entire document before parsing begins. If the CCSID to be used for processing the document is different from the CCSID of an RPG variable, CCSID conversion will be done on the data when it is assigned to the RPG variable.

- *best* indicates that the document should be processed in the CCSID that will best preserve the data in the document. If the document is in the job CCSID or an ASCII CCSID related to the job CCSID, the document will be processed in the job CCSID. Otherwise, the document will be processed in UCS-2 and the data will be converted to the job CCSID before it is assigned to variables with a data type other than UCS-2.
- *job* indicates that the document should be processed in the job CCSID. The data will be converted to UCS-2 when it is assigned to UCS-2 variables.
- *ucs2* indicates that the document should be processed in UCS-2. The data will be converted to the job CCSID when it is assigned to variables with a data type other than UCS-2.

When the XML document is in a file, the contents of the entire file may be converted to another CCSID before parsing begins.

The following table lists several files and their CCSIDs:

File	File CCSID	Related EBCDIC CCSID
file1.xml	37	37
file2.xml	1252	37
file3.xml	874	838
file4.xml	13488	(N/A, UCS-2)
file5.xml	1208	(N/A, UTF-8)

The following table shows the CCSID that would be used for processing these files for each value of the *ccsid* option, assuming the job CCSID is 37. An asterisk indicates that the file is converted to a different CCSID before processing:

File	CCSID Option Value		
	best	*job*	*ucs2*
file1.xml	37	37	13488*
file2.xml	37*	37*	13488*
file3.xml	13488*	37*	13488*
file4.xml	13488	37*	13488
file5.xml	13488*	37*	13488*

When the XML document is in a variable, the entire document may be converted to a different CCSID before parsing begins.

Given the following variable definitions:

```
D chrXml          S              100A
D ucs2Xml         S              100C
```

The following table shows the CCSID that would be used for processing these variables for each value of the "ccsid" option, assuming the job CCSID is 37. An asterisk indicates that the data in the variable is converted to a different CCSID before processing.

Variable	CCSID Option Value		
	best	*job*	*ucs2*
chrXml	37	37	13488
ucs2Xml	13488	37*	13488

path The *path* option specifies the path to the element as it appears in the XML document, with elements separated by forward slashes. For example, if this option is path=main/info/name, the parser will expect the document element to be "main", a child of "main" to be "info", and a child of "info" to be "name". If no element can be found, the operation will fail with status 00353 (XML does not match RPG variable).

Note: The value of the "allowmissing" option has no effect on this situation.

Note: The *path* option is required when %HANDLER is used to specify an array-handling procedure.

Default: When the *path* option is not specified, the search for the XML element matching the RPG variable depends on the type of the variable.

- For non-array variables, the outermost XML element is expected to have the same name as the RPG variable.
- For array variables, the outermost XML element is expected to have child elements with the same name as the RPG array variable. The outermost XML element can have any name.

Notes:

1. If the variable is a qualified subfield, only the name of the subfield is used in determining the path to the XML variable. For example, if the variable is DS.SUB1, the default is to expect the outermost XML element to be called "sub1".
2. The path specified by this option is case sensitive. It must be in the same case as the matching elements in the XML document unless the *case* option is also specified.

```
D info            DS
D   num                           5P 2

D xmlDoc          S           1000A    VARYING

D qualDs          DS                   QUALIFIED
D   subf                         10A

 /free
  // 1. Specifying a different name for the XML element
  xmlDoc = '<myinfo><num>123.45</num></myinfo>';
  xml-into info %XML(xmlDoc : 'path=myinfo');
  // num now has the value 123.45

  // 2. Neglecting to specify a different name for the XML
  //    element causes the operation to fail
  xmlDoc = '<myinfo><num>456.1</num></myinfo>';
  xml-into info %XML(xmlDoc');
  // The XML-INTO operation fails with status 00353 because the
  // document does not contain the "info" element

  // 3. Specifying that the XML element is not the outermost
  //    element in the document
  xmlDoc = '<data><info><num>-789</num></info></data>';
  xml-into info %XML(xmlDoc : 'path=data/info');
  // num now has the value -789

  // 4. Parsing into a subfield where the data structure is
  //    represented by the XML.  The full path to the "num"
  //    XML element must be specified.
  xmlDoc = '<data><info><num>.3</num></info></data>';
  xml-into num %XML(xmlDoc :
                  'path=data/info/num');
  // num now has the value .3

  // 5. Specifying the "path" option with XML from a file
  //    Assume file myfile.xml contains the following lines:
  //      <?xml version='1.0' ?>
  //      <data>
  //       <val>17</val>
  //      </data>
  xml-into num %XML('myfile.xml' : 'doc=file path=data/val');
  // num now has the value 17

  // 6. Specifying a qualified subfield without the "path"
  //    option.
  xmlDoc = '<subf>-987.65</subf>';
  xml-into qualDs.subf %XML(xmlDoc);
  // qualDs.subf now has the value '-987.65'

  // 7. Specifying a qualified subfield with the "path"
  //    option.
  //    Note that the default path for a qualified subfield
  //    is the subfield name; in this XML document, the
  //    XML element for the subfield is a child element
  //    of another XML element so the 'path' option must
  //    be specified, and it must include the names of all
  //    the ML elements in the path to the required XML
  //    element, including the XML element containing the
  //    data to set the variable.
  xmlDoc = '<qualds><subf>-987.65</subf></qualds>';
  xml-into qualDs.subf %XML(xmlDoc :
                   'path=qualds/subf);
  // qualDs.subf now has the value '-987.65'
```

Figure 400. Examples of the path option with non-array variables:

```
D loc             DS                    DIM(2)
D   city                         20A    VARYING
D   prov                          2A
D arr             S               5I 0  DIM(3)
D xmlDoc          S            1000A    VARYING

 /free
  // 1. Parsing an array from a string where the
  //    string contains array elements. The XML
  //    elements matching the RPG array elements
  //    are children of an XML element "outer".
  //    The "path" option is not needed because
  //    XML elements with the name "arr" are
  //    expected to be child elements of the
  //    outermost XML element.
  xmlDoc = '<outer>'
            + '<arr>3</arr>'
            + '<arr>4</arr>'
            + '<arr>-2</arr>'
            + '</outer> ;
  xml-into arr %XML(xmlDoc);
  // arr(1) = 3
  // arr(2) = 4
  // arr(3) = -2

  // 2. Parsing a DS array from a file where the
  //    file contains array elements with a
  //    container XML element.  The "path" option
  //    is not needed.  The name of the outermost element
  //    does not matter.
  //    Assume file myarray.xml contains the following lines:
  //      <locations>
  //       <loc><city>Saskatoon</city><prov>SK</prov></loc>
  //       <loc><city>Regina</city><prov>SK</prov></loc>
  //      </locations>
  xml-into loc %XML('myarray.xml' : 'doc=file');
  // loc(1).city = 'Saskatoon'  loc(2).city = 'Regina'
  // loc(1).prov = 'SK'         loc(2).prov = 'SK'

  // 3. Parsing a DS array where the XML elements have
  //    a different name from the array name.  The
  //    "path" option specifies the full path to the
  //    XML elements, including the container element
  //    "data".
  //    Assume file mydata.xml contains the following lines:
  //      <data>
  //       <where><city>Edmonton</city><prov>AB</prov></where>
  //       <where><city>Toronto</city><prov>ON</prov></where>
  //      </data>

  xmlfile = 'mydata.xml';
  xml-into loc %XML(xmlfile : 'path=data/where doc=file');
  // loc(1).city = 'Edmonton'   loc(2).city = 'Toronto'
  // loc(1).prov = 'AB'         loc(2).prov = 'ON'
```

Figure 401. Examples of the path option with array variables:

case (default *lower*)

The *case* option specifies the case expected for element and attribute names in the XML document when searching for XML data that matches the the RPG field names and the names in the *path* option. If the XML elements are not in the expected case, they will not be found, and the operation will fail with status 00353 (XML document does not match RPG variable) unless option 'allowmissing=yes' is specified.

- *lower* indicates that the XML element and attribute names matching the RPG variable names are in lower case.
- *upper* indicates that the XML element and attribute names matching the RPG variable names are in upper case.
- *any* indicates that the element and attribute names matching the RPG variable names are in unknown or mixed case. The XML element and attribute names will be converted to upper case before comparison to the upper-case RPG variable names.

```
D info           DS                    QUALIFIED
D   name                        10A
D   id_no                        5A
D xmlDoc         S            1000A    VARYING

 /free
  // 1. The XML document uses lowercase for element names and
  //    attributes.  The "case" option defaults to lowercase
  //    so it is not needed.
  xmlDoc = '<info><name>Jim</name><id_no>103</id_no></info>';
  xml-into info %XML(xmlDoc);
  // info.name = 'Jim       '
  // info.id_no = '103'

  // 2. The XML document uses uppercase for element names and
  //    attributes.  Option "case=upper" must be specified.
  xmlDoc = '<INFO><NAME>Bill</NAME><ID_NO>104</ID_NO></INFO>';
  xml-into info %XML(xmlDoc : 'case=upper');
  // info.name = 'Bill      '
  // info.id_no = '104'

  // 3. The XML document uses mixed case for element names and
  //    attributes.  Option "case=any" must be specified.
  xmlDoc = '<INFO><name>Tom</name>'
            + '<ID_NO>105</ID_NO></INFO>';
  xml-into info %XML(xmlDoc : 'case=any');
  // info.name = 'Tom       '
  // info.id_no = '104'

  // 4. The XML document uses mixed case for element names and
  //    attributes but the "case" option is not specified.
  xmlDoc = '<INFO><name>Tom</name>'
            + '<ID_NO>105</ID_NO></INFO>';
  xml-into info %XML(xmlDoc);
  // The XML-INTO operation fails with status 00353 because
  // it assumes the XML elements will have lowercase names.
```

Figure 402. Examples of the case option:

trim (default *all*)

The *trim* option specifies whether *whitespace* (blanks, newlines, tabs etc.) should be trimmed from text data before the data is assigned to RPG variables

- *all* indicates that before text content is assigned to the RPG character or UCS-2 variable, the following steps will be done:
  1. Leading and trailing whitespace will be trimmed completely from text content

   2. Strings of interior whitespace in the text content will be reduced to a single blank

- *none* indicates that no whitespace will be trimmed from text content. This option will have the best performance, but it should only be used if the whitespace is wanted, or if the XML data is known to contain no unwanted whitespace, or if the RPG program is going to handle the removal of the whitespace itself.

Notes:

1. Whitespace includes blank, tab, end-of-line, carriage-return, and line-feed.
2. This option applies only to XML data that is to be assigned to character and UCS-2 RPG variables. Trimming of whitespace is always done for other data types.
3. This option is mainly provided for XML data from files, but it also applies to XML data from a variable.
4. Whitespace between XML elements is always ignored. The trim option controls the whitespace within text content of elements and attributes.

```
D data            S            100A    VARYING
 //    Assume file data.xml contains the following lines:
 //      <text>
 //          line1
 //          line2
 //      </text>
 //
 //    Here is another view of this same file where
 //          '_' represents a blank
 //          'T' represents a tab
 //          'F' represents a line-feed
 //      <text>____F
 //      Tline1F
 //      ____line2F
 //      </text>F
 /free
  // 1. The default of "trim=all" is used.  Leading and
  //    trailing whitespace is removed.  Strings of
  //    internal whitespace is changed to a single blank.
  xml-into data %XML('data.xml' : 'doc=file');
  // data = 'line1 line2'

  // 2. Option "trim=none" is specified.  No whitespace
  //    is trimmed from text data.
  xml-into data %XML('data.xml' : 'doc=file trim=none');
  // The following line shows the value of data with the
  // line-feed and tab characters shown as ?.
  // data = '    ??line1?    line2?'
  // The following line shows the value of data with the
  // blanks, line-feed and tab characters shown as in the
  // second view of the document.
  // data = '____FTline1F____line2F'
```

Figure 403. Examples of the trim option:

allowmissing (default *no*)

For the situation where the XML document does not have sufficient XML elements or attributes for the subfields of an RPG data structure, you can use the *allowmissing* option to indicate whether this is considered an error. XML data is considered to be missing in the following circumstances:

- For an XML element matching an RPG data structure (including a data structure subfield), if the XML element does not have attributes or child elements for all RPG subfields.
- For XML data matching an array subfield of an RPG data structure, if the number of XML elements is less than the dimension of the RPG subfield array.

If expected XML data is not found, and 'allowmissing=yes' is not specified, the operation will fail with status 00353 (XML does not match RPG variable).

Tip: The *countprefix* option can also be used to handle the situation where the XML document might not have sufficient XML data for every subfield in the data structure.

To allow fewer array elements for the array specified on the XML-INTO operation, it is not necessary to specify 'allowmissing=yes'. If the XML document contains fewer elements than the RPG array, the operation will not fail. The "Number of XML Elements" subfield in positions 372 - 379 of the PSDS can be used to determine the number of elements successfully set by the operation.

- *no* indicates that XML data must be present for every subfield of a data structure (including subfields of data structure subfields), and XML data must be present for every element of every subfield array.
- *yes* indicates that when XML data is not present for every subfield and subfield array element, the operation will not fail. If a variable is specified as the first operand of XML-INTO, the unset subfields will hold the same value they held before the operation. If %HANDLER is specified as the first operand of XML-INTO, the unset subfields of the array passed to the handling procedure will have the default value for the type (zero for numeric values, *LOVAL for date values and so on).

```
D employee        DS                  QUALIFIED
D   name                       10A    VARYING
D   type                       10A

D empInfo3        DS                  QUALIFIED
D   emp                               LIKEDS(employee)
D                                     DIM(3)

D empInfo2        DS                  QUALIFIED
D   emp                               LIKEDS(employee)
D                                     DIM(2)

D empInfo4        DS                  QUALIFIED
D   emp                               LIKEDS(employee)
D                                     DIM(4)

 //  Assume file emp.xml contains the following lines:
 //    <employees>
 //     <emp><name>Jack</name><type>Normal</type></emp>
 //     <emp><name>Mary</name><type>Manager</type></emp>
 //     <emp><name>Sally</name><type>Normal</type></emp>
 //   </employees>

 /free
  // 1. The "empInfo3" data structure has an array "emp"
  //    with a dimension of 3.
  //    The "allowmissing" option is not required.
  //    The default of "allowmissing=no" can be used, since
  //    the XML document exactly matches the data structure.
  xml-into empInfo3 %XML('emp.xml' :
                         'doc=file path=employees');
  // empInfo3.emp(1)    .name = 'Jack'    .type = 'Normal'
  // empInfo3.emp(2)    .name = 'Mary'    .type = 'Manager'
  // empInfo3.emp(3)    .name = 'Sally'   .type = 'Normal'

  // 2. Option "allowmissing=no" may be specified, however.
  xml-into empInfo3 %XML('emp.xml' :
                         'doc=file ' +
                         'allowmissing=no path=employees');
  // empInfo3.emp(1)    .name = 'Jack'    .type = 'Normal'
  // empInfo3.emp(2)    .name = 'Mary'    .type = 'Manager'
  // empInfo3.emp(3)    .name = 'Sally'   .type = 'Normal'

  // 3. Option "allowmissing=yes" must be specified with
  //    data structure "empInfo4", since the XML document
  //    has only three "emp" XML elements, and the RPG "emp"
  //    array has four elements.
  xml-into empInfo4
           %XML('emp.xml' : 'doc=file ' +
                        'allowmissing=yes path=employees');
  // empInfo4.emp(1)    .name = 'Jack'    .type = 'Normal    '
  // empInfo4.emp(2)    .name = 'Mary'    .type = 'Manager   '
  // empInfo4.emp(3)    .name = 'Sally'   .type = 'Normal    '
  // empInfo4.emp(4)    .name = ''        .type = '          '

  // 4. Option "allowmissing" is not specified for data
  //    structure "empInfo4"
  xml-into empInfo4 %XML('emp.xml' :
                         'doc=file path=employees');
  // The XML-INTO operation fails with status 00353 because
  // the XML document does not have enough "emp" elements
  // for the RPG array.
```

Figure 404. Examples of the allowmissing option with insufficient data for subfield arrays:

```
D qualName        DS                    QUALIFIED
D   name                         10A
D   lib                          10A

D copyInfo        DS                    QUALIFIED
D   from                                LIKEDS(qualName)
D   to                                  LIKEDS(qualName)

 //    Assume file cpyA.xml contains the following lines:
 //      <?xml version='1.0' ?>
 //      <copyInfo>
 //       <to><name>MYFILE</name><lib>*LIBL</lib></to>
 //       <from name="MASTFILE" lib="CUSTLIB"></from>
 //      </copyInfo>

 //    Assume file cpyB.xml contains the following lines:
 //      <copyInfo>
 //       <from><name>MASTER</name><lib>PRODLIB</lib></from>
 //       <to><name>MYCOPY</name></to>
 //      </copyInfo>

 /free
  // 1. Data structure "copyInfo" has two subfields, "from"
  //    and "to".  Each of these subfields has two subfields
  //    "name" and "lib".  File "cpyA.xml" exactly matches
  //    the "copyInfo" structure, so the "allowmissing" option
  //    is not needed.
  xml-into copyInfo %XML('cpyA.xml' : 'doc=file');
  // copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
  // copyInfo.to    .name = 'MYFILE    ' .lib = '*LIBL     '

  // 2. File "cpyB.xml" is missing the "lib" subfield from
  //    the XML element "copyinfo.to".  Option
  //    "allowmissing=yes" must be specified to allow
  //    a subfield to be missing from the XML document.
  //    The copyInfo structure is cleared before the
  //    operation so the program can determine
  //    which subfields were not assigned any data.
  clear copyInfo;
  xml-into copyInfo %XML('cpyB.xml'
                   : 'doc=file allowmissing=yes');
  // copyInfo.from  .name = 'MASTER    ' .lib = 'PRODLIB   '
  // copyInfo.to    .name = 'MYCOPY    ' .lib = '          '
  //
  // The RPG program inspects the data to see if any subfields
  // have not been set.
  if copyInfo.from.lib = *blanks;
    copyInfo.from.lib = '*LIBL';
  endif;
  if copyInfo.to.lib = *blanks;
    copyInfo.to.lib = '*LIBL';
  endif;
```

Figure 405. Examples of the allowmissing option with insufficient data for all subfields:

allowextra (default *no*)

For the situation where the XML document has XML elements or attributes that are not needed for assignment to the subfields of an RPG data structure, you can use the *allowextra* option to indicate whether this is considered an error. XML data is considered to be extra in the following circumstances:

- For XML data matching an RPG data structure, if non-whitespace text content is found.

- For XML data matching an array subfield of an RPG data structure, if the number of XML elements is greater than the dimension of the RPG subfield array.
- For XML data matching a RPG scalar variable (neither data structure nor unindexed array), if the XML element contains child elements or attributes, other than the special formatting attributes allowed for some data types (see "Rules for transferring XML data to RPG variables" on page 881).

If unexpected XML data is found, and 'allowextra=yes' is not specified, the operation will fail with status 00353 (XML does not match RPG variable).

Warning: At any time, XML attributes for non-data-structure XML elements elements may be subject to interpretation by the RPG runtime. Currently, "fmt" and "adjust" are already being interpreted by the RPG runtime for some target data types. Support for other attributes may be added at any time, possibly even through PTFs. If an attribute is being ignored by option 'allowextra=yes', and that attribute becomes meaningful for the RPG runtime, it may affect the handling of the data.

- *no* indicates that the XML elements used to set the RPG variable or array elements must contain only the data necessary to set the variable.
- *yes* indicates that additional XML data will be ignored.

```
D employee        DS                  QUALIFIED
D   name                        10A    VARYING
D   type                        10A

D empInfo2        DS                  QUALIFIED
D   emp                               LIKEDS(employee)
D                                     DIM(2)

D empInfoAway     DS                  QUALIFIED
D   emp                               LIKEDS(employee)
D                                     DIM(2)
D   away                        10A    DIM(2)

 //  Assume file emp.xml contains the following lines:
 //    <employees>
 //     <emp><name>Jack</name><type>Normal</type></emp>
 //     <emp><name>Mary</name><type>Manager</type></emp>
 //     <emp><name>Sally</name><type>Normal</type></emp>
 //   </employees>

 /free

  // 1. Option "allowextra=yes" must be specified with
  //    data structure "empInfo2", since the XML document
  //    has three "emp" XML elements, and the RPG "emp"
  //    array only has two elements.
  xml-into empInfo2
          %XML('emp.xml'
             : 'doc=file allowextra=yes path=employees');
  // empInfo2.emp(1)    .name = 'Jack'    .type = 'Normal'
  // empInfo2.emp(2)    .name = 'Mary'    .type = 'Manager'

  // 2. Option "allowextra" is not specified for data structure
  //    "empInfo2"
  xml-into empInfo2
          %XML('emp.xml' : 'doc=file path=employees');
  // The XML-INTO operation fails with status 00353 because
  // the XML document has too many "emp" elements for the
  // RPG array.

  // 3. Structure "empInfoAway" requires 2 "emp" elements and
  //    2 "away" elements.  The XML document contains
  //    3 "emp" elements and zero "away" elements.
  //    Option "allowextra=yes allowmissing=yes" is specified,
  //    so the operation will succeed with any number of
  //    "emp" and "away" XML elements.  The extra "emp"
  //    element and missing "away" elements will be ignored.
  xml-into empInfoAway
          %XML('emp.xml' : 'allowextra=yes ' +
                           'allowmissing=yes ' +
                           'path=employees ' +
                           'doc=file');
  // empInfoSite.emp(1)  .name = 'Jack'    .type = 'Normal'
  // empInfoSite.emp(2)  .name = 'Mary'    .type = 'Manager'
  // empInfoSite.away(1) = ' '
  // empInfoSite.away(2) = ' '
```

Figure 406. Examples of the allowextra option with extra elements for a subfield array:

```
D qualName        DS                    QUALIFIED
D   name                          10A
D   lib                           10A

D copyInfo        DS                    QUALIFIED
D   from                                LIKEDS(qualName)
D   to                                  LIKEDS(qualName)

D copyInfo3       DS                    QUALIFIED
D   from                                LIKEDS(qualName)
D   to                                  LIKEDS(qualName)
D   create                         1N

 //    Assume file cpyA.xml contains the following lines:
 //      <copyInfo>
 //       <to><name>MYFILE</name><lib>*LIBL</lib></to>
 //       <from name="MASTFILE" lib="CUSTLIB"></from>
 //      </copyInfo>

 //    Assume file cpyC.xml contains the following lines:
 //      <copyinfo errors="tolerate">
 //       <to><name>MYFILE</name><lib>MYLIB</lib></to>
 //       <from><name>MASTFILE</name><lib>CUSTLIB</lib></from>
 //       <to><name>MYFILE2</name></to>
 //      </copyinfo>

 //    Assume file cpyD.xml contains the following lines:
 //      <copyinfo to="MYLIB/MYFILE">
 //       <from><name>MASTFILE</name><lib>CUSTLIB</lib></from>
 //      </copyinfo>

 /free
  // 1. Data structure "copyInfo" has two subfields, "from"
  //    and "to".  Each of these subfields has two subfields
  //    "name" and "lib".  File "cpyA.xml" exactly matches
  //    the "copyInfo" structure, so the "allowextra" option
  //    is not needed, since "allowextra" defaults to "yes".
```

Figure 407. Examples of the allowextra option with XML data not corresponding to RPG subfields: (Part 1 of 2)

```
xml-into copyInfo %XML('cpyA.xml' : 'doc=file');
// copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo.to    .name = 'MYFILE    ' .lib = '*LIBL     '

// 2. File "cpyC.xml" has an XML attribute for the
//    for the XML element "copyinfo" that does not
//    match an RPG subfield.  It also has the
//    "to" subfield specified more than once.  Option
//    "allowextra=yes" must be specified to allow
//    extra subfields in the XML document.
//    The extra XML data will be ignored.
xml-into copyInfo
         %XML('cpyC.xml' : 'doc=file allowextra=yes');
// copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo.to    .name = 'MYFILE    ' .lib = 'MYLIB     '

// 3. Data structure copyInfo3 has a subfield
//    "create" that does not appear file "cpyC.xml".
//    "cpyC.xml" has both missing and extra subfields
//    for data structure "copyInfo3".
//    Options "allowextra=yes allowmissing=yes" must
//    both be specified.
//    The extra subfields will be ignored and the
//    missing subfield will retain its original value.
clear copyInfo3;
xml-into copyInfo3
         %XML('cpyC.xml' : 'allowextra=yes ' +
                           'allowmissing=yes ' +
                           'doc=file' +
                           'path=copyinfo');
// copyInfo3.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo3.to    .name = 'MYFILE    ' .lib = 'MYLIB     '
// copyInfo3.create = '0' (from the CLEAR operation)

// 4. File "cpyD.xml" has an XML element "copyInfo"
//    with an attribute "to".  Subfields can be specified
//    by attributes only when the subfield is neither
//    an array nor a data structure.
xml-into copyInfo %XML('cpyC.xml' : 'doc=file');
// The XML-INTO operation fails because the "to" attribute
// is not expected, and because the "to" XML element is
// not found.

// 5. Options "allowextra=yes allowmissing=yes" are
//    specified, allowing the extra "to" attribute to be
//    ignored and the missing "to" element to be tolerated.
//    The "to" subfield is not changed by the XML-INTO
//    operation.
copyInfo.to.name = '*UNSET*';
copyInfo.to.lib = '*UNSET*';
xml-into copyInfo %XML('cpyD.xml' : 'doc=file ' +
                  'allowextra=yes allowmissing=yes');
// copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo.to    .name = '*UNSET*   ' .lib = '*UNSET*   '
```

Figure 407. Examples of the allowextra option with XML data not corresponding to RPG subfields: (Part 2 of 2)

```
D part            DS
D   size                          10A

 //    Assume file part.xml contains the following lines:
 //      <?xml version='1.0' ?>
 //       <part>light bulb<size>medium</size></part>

 /free
  // 1. "part" is a data structure.  The XML file
  //    part.xml has an element called "part" with
  //    both element and text children
  xml-into part %XML('part.xml' : 'doc=file');
  // The XML-INTO operation fails because the "part" XML
  // element has text content ("light bulb"),
  // and the "allowextra" option defaults to "no".

  // 2. "allowextra=yes" is specified, allowing the
  //    text content to be ignored
  xml-into part %XML('part.xml' : 'doc=file allowextra=yes');
  // size = 'medium'
```

Figure 408. Examples of the allowextra option with unexpected text content for a data structure:

```
D text            S            200A    VARYING

D order           DS                   QUALIFIED
D   part                        25A    VARYING
D   quantity                    10I 0

 //    Assume file txt.xml contains the following lines:
 //      <?xml version='1.0' ?>
 //      <text><word>Hello</word><word>World</word></text>

 //    Assume file ord.xml contains the following lines:
 //      <?xml version='1.0' ?>
 //      <order>
 //       <part>Jack in a box<discount>yes</discount></part>
 //       <quantity multiplier="10">2</quantity>
 //      </order>

 /free
  // 1. "text" is a standalone variable.  The XML file
  //    txt.xml has an element called "text" with two
  //    child elements called "word".
  xml-into text %XML('txt.xml' : 'doc=file');
  // The XML-INTO operation fails because the "text" XML
  // element has child elements, and the "allowextra"
  // option defaults to "no".

  // 2. "allowextra=yes" is specified.  The child elements
  //    are ignored.
  xml-into text %XML('txt.xml' : 'allowextra=yes doc=file';
  // The XML-INTO operation succeeds, but since the
  // only content for the "text" XML element is the child
  // XML elements, no data is available for RPG field "text".
  //   text = ''

  // 3. "order" is a data structure with two subfields
  //    which are not themselves data structures.
  //    The XML elements representing the subfields
  //    should not have child elements or attributes, but the
  //    "part" XML element does have one child, "discount",
  //    and the "quantity" XML element has an attribute
  //    "multiplier".  Option "allowextra=yes" is specified,
  //    so the "discount" element and "multiplier" attribute
  //    are ignored.
xml-into order %XML('ord.xml' : 'doc=file allowextra=yes');
  // order.part = "Jack in a box"
  // order.quantity = 2
```

Figure 409. Examples of the allowextra option with unexpected non-text content for a scalar variable or subfield:

datasubf

The *datasubf* option specifies the name of the extra scalar subfield used to handle the situation where there is text data for an XML element that matches an RPG data structure.

For example, if this option is specified as datasubf=txt, and an RPG data structure has a scalar subfield with name *txt*, then that subfield will receive the text data for the XML element matching the data structure.

Default: When the *datasubf* option is not specified, XML elements matching RPG data structures cannot have text data. Text data can only be associated with the subfields of the data structure.

Notes:

1. When an RPG data structure has a scalar subfield whose name is specified by the *datasubf* option, the following rules apply:
   - If the matching XML element has text data, that text data will be assigned to the scalar subfield.
   - The values for all the other subfields of the data structure must be set by XML attributes. Therefore, the XML element cannot have any child elements, and the other subfields of the data structure must all be scalar subfields.
   - The XML element matching the data structure cannot have an XML attribute or child XML element with the same name as the *datasubf* option.
   - If the XML element does not have any text data, the *datasubf* subfield will be set to an empty value. If the datatype of the subfield does not support the empty value, for example numeric and date types, assigning the subfield will result in an exception.
2. When an RPG data structure does not have a scalar subfield whose name is specified by the *datasubf* option, the *datasubf* option is ignored for that data structure. The XML element matching the RPG data structure cannot have text data.
3. When an RPG data structure has an array or data structure subfield whose name is the same as the name specified by the datasubf option, the datasubf option is ignored for that data structure. The XML element matching the RPG data structure cannot have text data.
4. A complex RPG data structure may have many data structure subfields. The *datasubf* option is considered separately for each data structure subfield. The XML data for one data structure subfield might require the datasubf option for the XML-INTO operation to complete successfully, while another data structure subfield might not require it.
5. A datasubf subfield cannot be the same as a countprefix subfield. For example, if countprefix=num_ was specified, and the data structure has subfields arr and num_arr, then num_arr is a countprefix subfield. Option datasubf=num_arr cannot also be specified for this data structure.

```
D customer        ds                  qualified
D    id                          10a
D    value                      100a   varying

D order           ds                  qualified
D    id                          10a
D    type                        10a

D customers       ds                  qualified
D    customer                         likeds(customer) dim(2)

D orderinfo       ds                  qualified
D    customer                         likeds(customer)
D    order                            likeds(order)
 /free

  // 1. The datasubf option specifies the "value" subfield.
  //
  //    Assume file customer1.xml contains the following
  //    <customer id="A34R27K">John Smith</customer>

  //    When XML-INTO encounters "John Smith", it is
  //    processing the "customer" data structure.  It
  //    finds that the "customer" data structure has a
  //    subfield called "value", so it uses that subfield
  //    for the "John Smith" data.
  xml-into customer %xml('customer1.xml'
                      : 'doc=file datasubf=value');
  // customer.id = "A34R27K"
  // customer.value = "John Smith"

  // 2. The datasubf option is not specified.
  //
  //    Assume file customer2.xml contains the following
  //    <customer id="A34R27K">John Smith</customer>

  //    When XML-INTO encounters "John Smith", it is
  //    processing the "customer" data structure.  XML-INTO
  //    does not normally support having data for a data
  //    structure, so the XML-INTO operation fails due to
  //    extra XML data.
  xml-into(e) customer %xml('customer2.xml'
                         : 'doc=file');
  // %error = *on

  // 3. The XML document has an ordinary XML element
  //    whose name is the same as the datasubf option.
  //
  //    Assume file customer3.xml contains the following
  //    <customer id="A34R27K">
  //       <value>John Smith</value>
  //    </customer>

  // The datasubf option is not specified.
  // The XML document has an ordinary XML element called
  // "value", so the "value" subfield of the "customer"
  // data structure is filled in the usual way.
  // The datasubf option is not needed.
  xml-into customer %xml('customer3.xml' : 'doc=file');
  // customer.id = "A34R27K"
  // customer.value = "John Smith"

  // The datasubf=value option is specified.
  // The XML document has an ordinary XML element called
  // "value".  The XML-INTO operation fails because a
  // scalar subfield with the name of the datasubf option
  // cannot be filled by an XML attribute or an XML element.
  xml-into(e) customer %xml('customer3.xml'
                         : 'doc=file datasubf=value');
  // %error = *on

  // 4. For a complex data structure, the datasubf option
```

countprefix

The *countprefix* option specifies the prefix for the subfields that can receive the number of elements that were set by an XML-INTO operation for a subfield array. The name of the count subfield is formed by adding the array name to the countprefix value. For example, if a data structure has a subfield array meeting.attendees, and countprefix=num was specified, the XML-INTO operation would set meeting.numattendees to the actual number of elements of the meeting.attendees array that were set by the XML-INTO operation. In the subsequent discussion of the countprefix option, subfield meeting.numattendees is referred to as the *countprefix subfield* and meeting.attendees is referred to as the *counted subfield*.

The processing for the countprefix option is done after the XML data for a data structure or data structure subfield has been parsed.

Notes:

1. A countprefix subfield must be numeric, and it must be scalar; that is, it cannot be an array or a data structure. If a subfield has a countprefix name, but is not numeric or scalar, that subfield will be processed normally; it will not be considered to be a countprefix subfield.
2. A counted subfield can be any type of subfield; it is not required to be an array. If a counted subfield is not an array, its countprefix subfield will be set to 0 (zero) if there is no XML data to set the subfield, and it will be set to 1 (one) if there is XML data to set it.
3. When a subfield is counted by a countprefix subfield, the allowmissing option is not considered for that subfield. Option allowmissing=yes is implied for all subfields that are counted by a countprefix subfield.
4. If there is too much XML data for a subfield, the countprefix subfield will only reflect the number of array elements that were actually set by the XML-INTO operation. For example, if array arr has ten elements, and there is XML data for eleven elements, the countprefix subfield for arr would have the value 10.
5. If the XML-INTO operation ends in error, the countprefix subfields may not reflect the exact number of RPG subfields that were updated by the XML-INTO operation. The countprefix processing is done after the XML data for each data structure or data structure subfield has been parsed; if an error occurs during parsing, or during the countprefix processing, the countprefix processing would not be completed.
6. A countprefix subfield is not considered to be countable. For example, if countprefix=num_ was specified, and the data structure has subfields arr, num_arr and num_num_arr, then num_arr would be considered a countprefix subfield for array arr, but num_num_arr would not be considered a countprefix subfield for num_arr.
7. A countprefix subfield cannot be explicitly set by XML data. Any XML attributes or XML elements that set a countprefix subfield are considered to be extra.
8. A countprefix subfield cannot be the same as a datasubf subfield. For example, if countprefix=num_ was specified, and the data structure has subfields arr and num_arr, then num_arr is a countprefix subfield. Option datasubf=num_arr cannot also be specified for this data structure.

```
D attendee_type...
D                 DS                  qualified template
D   name                        20a   varying
D   phone                        4s 0

D meeting         DS                  qualified
D   location                    20a   varying
D   attendee                          likeds(attendee_type)
D                                     dim(100)
D   numAttendee...
D                               10i 0

D email           DS                  qualified
D   to                          40a   varying
D   cc                          40a   varying
D   from                        40a   varying
D   countCc                      5i 0
D   subject                    100a   varying
D   countSubject                 5i 0
D   body                      1000a   varying

D order1          DS                  qualified
D   numpart                     10i 0
D   part                        20a   varying dim(100)

D order2          DS                  qualified
D   numpart                     10i 0
D   part                        20a   varying dim(100)
D   countpart                   10i 0

1. File meeting123.xml:
<meeting>
   <location>Room 7a</location>
   <attendee name="Jim" phone="1234"/>
   <attendee name="Mary" phone="2345"/>
   <attendee name="Abel" phone="6213"/>
</meeting>

 // a. The countprefix option specifies the "num" prefix.
 //
 // The XML-INTO operation sets countprefix subfield
 // "numAttendee" to 3, the number of "attendee" subfields
 // set by the operation.  It is not necessary to
 // specify option allowmissing=yes, because the
 // presence of the countprefix subfield for array
 // attendee implictly allows missing XML data for
 // that particular array.
 xml-into meeting %xml('meeting123.xml'
                 : 'doc=file countprefix=num');
 // meeting.attendee(1):  name='Jim'   phone=1234
 // meeting.attendee(2):  name='Mary'  phone=2345
 // meeting.attendee(3):  name='Abel'  phone=6213
 // meeting.numAttendee = 3
 for i = 1 to meeting.numAttendee;
    // process meeting.attendee(i)
 endfor;

 // b. The countprefix subfield is not specified.
 //
 // The XML-INTO operation fails because there is
 // insufficient XML data for array "attendee", and
 // there is no XML data at all for "numAttendee"
 xml-into(e) meeting %xml('meeting123.xml'
                 : 'doc=file');
 // %error is set on

2. File email456.txt:
<email to="jack@anywhere.com" from="jill@anywhere.com">
   <subject>The hill</subject>
   <body>How are you feeling after your fall?</body>
</email>
```

Expected format of XML data

The structure of the XML elements is expected to match the structure of the RPG variable.

- The XML element matching the RPG variable can be at any nesting level of the XML document, but the *path* option must be specified if the XML element is not at the assumed nesting level of the document. The following assumptions are made when the *path* option is not specified.
  - For non-array variables (including table names and multiple occurrence data structures, the document element (the outermost XML element) is assumed to be the XML element matching the RPG variable. If the name of the outermost XML element is not the same as the name of the RPG variable, the *path* option must be used to specify the XML element to be used.
  - For array variables, direct children of the document element (the outermost XML element) are assumed to be the XML elements matching the RPG variable
- XML elements matching an RPG subfield can be
  - XML attributes of the XML element matching the RPG subfield's parent data structure (only for subfields that are not themselves data structures)
  - direct child XML elements of the XML element matching the data structure containing the subfield
- XML elements matching RPG arrays must be children of the same XML parent. It is not required that these child elements appear together in the XML document; they may be interleaved with other elements.

Note: XML processing instructions are ignored by XML-INTO. Processing instructions are in the form

```
<?targetname data value ?>
```

Scalar variable

```
D libname      S            10A
 /free
     XML-INTO libname %XML(xmldoc : option)
```

Sample XML for XML-INTO libname	*path* **option**
<libname>*data*</libname>	
<library>*data*</library>	'path=library'
<info><library>*data*</library></info>	'path=info/library'

Array element

```
D sites        S            25A   DIM(3)
 /free
     XML-INTO sites(n) %XML(xmldoc : option)
```

Sample XML for XML-INTO sites	*path* **option**
<sites>*data*</sites>	blank
<custsites>*data*</custsites>	'path=custsites'
<info><sites>*data*</sites></info>	'path=info/sites'

Table name

```
D tabname      S            10A   DIM(5)
 /free
     XML-INTO tabname %XML(xmldoc : opts)
```

Sample XML for XML-INTO tabname	*path* option
<tabname>*data*</tabname>	blank
<library>*data*</library>	'path=library'
<info><library>*data*</library></info>	'path=info/library'

Simple data structure or multiple-occurrence data structure

Note: The XML data in the examples show line breaks and indentation for clarity only. The XML data may be formatted in any convenient way.

OR

```
D pgm           DS
D   name                    10A
D   lib                     10A
```

Figure 412.

```
D pgm           DS                 OCCURS(5)
D   name                    10A
D   lib                     10A
 /free
      XML-INTO pgm %XML(xmldoc : option)
```

Figure 413.

Sample XML for XML-INTO pgm	*path* option

`<pgm>`
  `<name>`*data*`</name>`
  `<lib>`*data*`</lib>`
`</pgm>`
	blank

`<program>`
  `<name>`*data*`</name>`
  `<lib>`*data*`</lib>`
`</program>`
	'path=program'

`<api>`
  `<program>`
    `<name>`*data*`</name>`
    `<lib>`*data*`</lib>`
  `</program>`
`</api>`
	'path=api/program'

Note: The subfield information can come from XML elements or XML attributes. The following show other valid ways to specify the XML for the subfields of the data structure. The designer of the XML document can use either attributes or elements freely when representing the XML data for a scalar subfield.

```
 <pgm name="data" lib="data"/>
OR
 <pgm name="data">
   <lib>data</lib>
 </pgm>
```

Array of scalar type

```
D sites         S              25A   DIM(3)
 /free
      XML-INTO sites %XML(xmldoc : option)
```

Sample XML for XML-INTO sites	*path* option
<anything> <sites>*data*</sites> <sites>*data*</sites> <sites>*data*</sites> </anything>	*blank*
<info> <custsites>*data*</custsites> <custsites>*data*</custsites> <custsites>*data*</custsites> </info>	'path=info/custsites'

Array of data structures

```
D pgm          DS                    DIM(3) QUALIFIED
D   name                     10A
D   lib                      10A
 /free
      XML-INTO pgm %XML(xmldoc : option)
```

Sample XML for XML-INTO pgm	*path* option
<anything> <pgm name="name1" lib="lib1"/> <pgm><name>name2</name> <lib>lib2</lib></pgm> <pgm lib="lib3"><name>name3</pgm> </anything>	*blank*
<programs> <pgm name="name1" lib="lib1"/> <pgm><name>name2</name> <lib>lib2</lib></pgm> <pgm lib="lib3"><name>name3</pgm> </programs>	'path=programs/pgm'

Note: The three "pgm" XML elements have the name and lib information specified in various combinations of XML elements and XML attributes. The designer of the XML document can use either attributes or elements freely when representing the XML data for a scalar subfield.

Complex data structure

```
D qualname     DS                  QUALIFIED
D   name                     10A
D   lib                      10A
D dtaaraInfo   DS                  QUALIFIED
D   dtaara                         LIKEDS(qualname)
D   type                     10I 0
D   value                   100a
 /free
      XML-INTO dtaaraInfo %XML(xmldoc : option)
```

Sample XML for XML-INTO dtaaraInfo	*path* option
<dtaarainfo> <dtaara> <name>*data*</name> <lib>*data*</lib> </dtaara> <type>*data*</type> <value>*data*</value> </dtaarainfo>	*blank*

<table>
<tr><th>Sample XML for XML-INTO dtaaraInfo</th><th>path option</th></tr>
<tr><td><pre>
<sys>
  <obj>
    <dta>
       <dtaara>
         <name>data</name>
         <lib>data</lib>
       </dtaara>
       <type>data</type>
       <value>data</value>
    </dta>
  </obj>
 </sys>
</pre></td><td>'path=sys/obj/dta'</td></tr>
</table>

Handler procedure with array of data structures

```
D myCommArea   DS
D   total                 20u 0
D custType     DS                qualified
D   name                  50a    varying
D   id_no                 10i 0
D   city                  20a
D custHdlr     PR
D   commArea                     likeds(myCommArea)
D   custinfo                     likeds(custType) dim(5)
D   numElems              10u 0  const
 /free
      XML-INTO %HANDLER(custHdlr : myCommArea) %XML(xmldoc : option)
```

Note: The *path* option is required when %HANDLER is specified.

<table>
<tr><th>Sample XML for XML-INTO %HANDLER(custHdlr:x)</th><th>path option</th></tr>
<tr><td><pre>
<info>
  <cust>
     <name>data</name>
     <id_no>data</id_no>
     <city>data</city>
  </cust>
  <cust>
     <name>data</name>
     <id_no>data</id_no>
     <city>data</city>
  </cust>
  :
  :
  <cust>
     <name>data</name>
     <id_no>data</id_no>
     <city>data</city>
  </cust>
</info>
</pre></td><td>'path=info/cust'</td></tr>
</table>

Handler procedure with array of scalar types

```
D myCommArea   DS
D   total                 20u 0
D nameHdlr     PR
D   commArea                     likeds(myCommArea)
D   names                 10a    dim(5)
D   numNames              10u 0  const
 /free
      XML-INTO %HANDLER(nameHdlr : myCommArea) %XML(xmldoc : option)
```

Note: The *path* option is required when %HANDLER is specified.

Sample XML for XML-INTO %HANDLER(nameHdlr:x)	*path* **option**
<info> <name>*data*</name> <name>*data*</name> <name>*data*</name> <name>*data*</name> : : <name>*data*</name> <name>*data*</name> </info>	'path=info/names'

Rules for transferring XML data to RPG variables

- For integer, unsigned, decimal (packed, zoned, binary) and float fields , the data will be transferred using the same rules as RPG uses for %INT, %UNS, %DEC, %FLOAT for respectively. %INTH, %UNSH and %DECH will be used if the Half-Adjust operation extender is specified on the XML-INTO operation code.
- For date, time and timestamp fields, the data will be transferred using the same rules as RPG uses for %DATE, %TIME and %TIMESTAMP respectively. The format defaults to *ISO with separators. The format may be specified by an attribute *fmt* in the element. The value of the attribute must be one of the valid formats for the respective built-in function; the leading asterisk is optional. For formats that allow more than one separator in RPG, the separator defaults to the RPG default separator for the format. For example, for a date field, the following XML fragments are valid:

  ```
  <myDate fmt="DMY/">25/12/04</myDate>   <!-- 2004-12-25 -->
  <myDate fmt="Dmy">25.12.04</myDate>    <!-- 2004-12-25 -->
  <myDate fmt="*cymd0">0971123</myDate>  <!-- 1997-11-23 -->
  ```
- For indicator, character and UCS-2 fields, data will be transferred with appropriate CCSID conversion if necessary. Fixed-length fields will be assigned left-adjusted by default. The adjustment can be specified by an attribute *adjust* in the element, with a value of either "left" or "right". For example, if the RPG variable *data* is 10 bytes long, the following XML data will cause the value of DATA to be set to 'ƀƀƀƀƀabcde'.

  ```
  <data adjust="right">abcde</data>
  ```
- For graphic fields, data will be transferred using the same rules as the %GRAPH built-in function, with appropriate CCSID conversion if necessary. Fixed-length fields will be assigned left-adjusted by default. The adjustment can be specified by an attribute *adjust* in the element, with a value of either "left" or "right".
- Pointer and procedure-pointer subfields are not supported, and are ignored by the XML-INTO operation.
- The special attributes *fmt* and *adjust* will be treated as ordinary attributes if they are not relevant to the assignment of the matching variable, or if the value of the attribute is not valid. For example, the following XML attributes would be treated as ordinary XML attributes:

  'fmt="abc"'
  : "abc" is not a valid format

  'adjust=yes'
  : "yes" is not a valid value for the "adjust" attribute

  'fmt="mdy/"', if specified for a numeric field

'adjust=right', if specified for a varying-length field

- The attributes *fmt* and *adjust* and their values must be specified in the case specified by the *case* option. The following table shows valid examples of the attributes for each value of the *case* option.

case **option**	*fmt*, **example "*MDY/"**	*adjust*, **example "right"**
not specified	`fmt="mdy/"` `fmt="*mdy/"`	adjust="right"
'case=lower'	`fmt="mdy/"` `fmt="*mdy/"`	adjust="right"
'case=upper'	`fmt="MDY/"` `fmt="*MDY/"`	ADJUST="RIGHT"

'case=any'	`Fmt="Mdy/"`
	`FMT="*mDY/"`
	`and so on`
		`Adjust="Right"`
		`adjust="RIGHT"`
		`and so on`

Examples of the XML-INTO operation

```
D qualName       DS                   QUALIFIED
D   name                        10A
D   lib                         10A

D copyInfo       DS                   QUALIFIED
D   from                              LIKEDS(qualName)
D   to                                LIKEDS(qualName)

D toName         S              10A   VARYING

 //  Assume file cpyA.xml contains the following lines:
 //    <copyinfo>
 //     <to><name>MYFILE</name><lib>*LIBL</lib></to>
 //     <from name="MASTFILE" lib="CUSTLIB"></from>
 //    </copyinfo>
 /free
       // Data structure "copyInfo" has two subfields, "from"
       // and "to".  Each of these subfields has two subfields
       // "name" and "lib".
       xml-into copyInfo %XML('cpyA.xml' : 'doc=file');
       // copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
       // copyInfo.to    .name = 'MYFILE    ' .lib = '*LIBL     '

       // Parse the "copyinfo/to/name" information into variable
       // "toName".  Use the "path" option to specify the location
       // of this information in the XML document.
       xml-into toName %XML('cpyA.xml'
                        : 'doc=file path=copyinfo/to/name';
       // toName = 'MYFILE'
```

Figure 414. Parsing directly into a variable from a file

```
D info            DS
D   name                          10A
D   val                            5I 0  DIM(2)
D xmlFragment     S             1000A    VARYING
D opts            S               20A    INZ('doc=string')
D dateVal         S               10A    INZ('12/25/04')
D format          S                4A    INZ('mdy/')
D mydate          S                 D    DATFMT(*ISO)

 /free

  // 1. Parsing into a data structure containing an array
  xmlFragment = '<info><name>Jill</name>'
              + '<val>10</val><val>-5</val></info>';
  xml-into info %XML(xmlFragment);
  // info now has the value
  //    name = 'Jill'
  //    val(1) = 10
  //    val(2) = -5

  // 2. Parsing into a date.  The "fmt" XML attribute indicates the
  //    format of the XML date.
  xmlFragment = '<mydate fmt="' + format + '">'
              + dateVal + '</mydate>';
  xml-into mydate %XML(xmlFragment);
  // xmlFragment = '<mydate fmt="mdy">12/25/04</mydate>'
  // mydate = 2004-12-25
```

Figure 415. Parsing directly into a variable from a string variable

```
// DDS for "MYFILE"
// A          R PARTREC
// A            ID           10P 0
// A            QTY          10P 0
// A            COST          7P 2

// XML data in "partData.xml"
//  <parts>
//   <part><qty>100</qty><id>13</id><cost>12.03</cost></part>
//   <part><qty>9</qty><id>14</id><cost>3.50</cost></part>
//   ...
//   <part><qty>0</qty><id>254</id><cost>1.98</cost></part>
//  </records>

Fmyfile    if   e             disk
D options         S            100A
D allOk           S              N

D partHandler     PR            10I 0
D   ok                           N
D   parts                             LIKEREC(partrec) DIM(10)
D   numRecs                     10U 0 VALUE

:
:
 /free
   // Initiating the parsing
   options = 'doc=file path=parts/part';
   allOk = *ON;
   xml-into %HANDLER(partHandler : allOk)
            %XML('partData.xml' : options);
   // Check if the operation wrote the data
   // successfully
   if not allOk;
      // some output error occurred
   endif;
 /end-free

:
:
 // The procedure to receive the data from up to 10
 // XML elements at a time.  The first call to the
 // this procedure would be passed the following data
 // in the "parts" parameter:
 //   parts(1) .id = 13   .qty = 100  .cost = 12.03
 //   parts(2) .id = 14   .qty = 9    .cost = 3.50
 //   ...
 // If there were more than 10 "part" child elements in
 // the XML file, this procedure would be called more
 // than once.

P partHandler     B
D                 PI            10I 0
D   ok                          1N
D   parts                             LIKEREC(partrec) DIM(10)
D   numRecs                     10U 0 VALUE

D i               S             10I 0
 * xmlRecNum is a static variable, so it will hold its
 * value across calls to this procedure.
 * Note: Another way of storing this information would be to
 * pass it as part of the first parameter; in that
 * case the first parameter would be a data structure
 * with two subfields: ok and xmlRecNum
```

Figure 416. Parsing an unknown number of XML elements using a handling procedure (Part 1 of 2)

```
D xmlRecNum       S             10I 0 STATIC INZ(0)
 /free
   for i = 1 to numRecs;
      xmlRecNum = xmlRecNum + 1;
      write(e) partRec parts(i);
      // Parameter "ok" was passed as the second parameter
      // for the %HANDLER built-in function for the XML-INTO
      // operation.  The procedure doing the XML-INTO
      // operation can check this after the operation to
      // see if all the data was written successfully.
      if %error;
        // log information about the error
        logOutputError (xmlRecNum : parts(i));
        ok = *OFF;
      endif;
   endfor;

   // continue parsing
   return 0;
 /end-free
P                 E
```

Figure 416. Parsing an unknown number of XML elements using a handling procedure (Part 2 of 2)

For more information about XML operations, see "XML Operations" on page 475.

XML-SAX (Parse an XML Document)

Free-Form Syntax	XML-SAX{(E)} %HANDLER(*handlerProc* : *commArea*) %XML(*xmlDoc* {: *options* });

Code	Factor 1	Extended Factor 2
XML-SAX{(E)}		%HANDLER(*handlerProc* : *commArea*) %XML(*xmlDoc* {: *options* })

Tip: If you are not familiar with the basic concepts of XML and of processing XML documents, you may find it helpful to read the "Processing XML Documents" section in *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* before reading further in this section.

XML-SAX initiates a SAX parse for an XML document. The XML-SAX operation code begins by calling an XML parser which begins to parse the document. When an event occurs such as the parser finding the start of an element, finding an attribute name, finding the end of an element and so on, the parser calls the handling procedure *handlerProc* with parameters describing the event. When the handling procedure returns, the parser continues to parse until it finds the next event and calls the handling procedure again. When the parser has finished parsing the document, control passes to the statement following the XML-SAX operation.

The first operand must be the %HANDLER built-in function; *handlerProc* is a prototype name that specifies the procedure to be called to handle the SAX events and *commArea* is the communication-area parameter to be passed by the parser to the handling procedure. The communication-area parameter must be the same type as the first prototyped parameter of the handling procedure. It provides a way for the procedure specifying the XML-SAX operation code to communicate with the handling procedure, and for the handling procedure to save information related to the parse from one event to the next. See "%HANDLER (handlingProcedure : communicationArea)" on page 539 for more information on %HANDLER.

The second operand must be the %XML built-in function, identifying the XML document to be parsed and the options controlling the way the parsing is done. See "%XML (xmlDocument {:options})" on page 604 for more information on %XML.

Operation extender E can be specified to handle the following status codes:

00351 Error in XML parsing

00352 Invalid XML option

00354 Error preparing for XML parsing

For status 00351, the return code from the parser will be placed in the subfield "External return code" in positions 368-371 of the PSDS. This subfield will be set to zero at the beginning of the operation and set to the value returned by the parser at the end of the operation. This subfield is relevant only in a module that has an XML-SAX operation. SAX event-handling procedures receive the information from the parser as parameters.

The event-handling procedure will not be called if an exception occurs before parsing begins. For example, if the specified file is not found, the operation will end immediately with status 00354 and the event-handling procedure will never get control.

If an error occurs during parsing, the handling procedure will be called with a *XML_EXCEPTION event, and when the handling procedure returns, parsing will end and the XML-SAX operation will fail with status code 00351. The return code from the parser will be placed in the "External return code" subfield in positions 368 - 371 of the PSDS.

If an unknown, invalid or unrelated option is found in the %XML options string, XML-SAX will fail with status code 00352. The External return code subfield in positions 368 - 371 of the PSDS will not be updated from the initial value of zero, set when the operation begins.

%XML options for the XML-SAX operation code

doc (default *string*)

The *doc* option indicates what the source operand of %XML contains.

- *string* indicates that the source operand contains XML data
- *file* indicates that the source operand contains an IFS file name

```
// In the following example, the first parameter
// of %XML is the name of a file.  Option
// "doc=file" must be specified.
ifsfile = 'myfile.xml';
opt = 'doc=file';
XML-SAX %handler(hdlr:comm) %XML(ifsfile : opt);

// In the following example, the first parameter
// of %XML is an XML document.  Since the "doc"
// option defaults to "string", no options are
// necessary.
xmldata = '<data><num>3</num></data>';
XML-SAX %handler(hdlr:comm) %XML(xmldata);
```

Figure 417. Example of the doc option:

ccsid (default *job*)

The *ccsid* option specifies the CCSID that the XML data should be returned in.

- *job* indicates that the XML parser should return data in the job CCSID. This is the CCSID that the RPG compiler uses for character data in the program.
- *ucs2* indicates that the XML parser should return data in the UCS-2 CCSID of the module.
- *numeric value* indicates that the XML parser should return the data in the specified CCSID. In this case, it is up to the RPG programmer to ensure that the data is handled correctly within the RPG program. The RPG compiler will assume that character data is in the job CCSID.

```
// In the following example, the data is to be
// returned in the job ccsid.  Even though the
// default for the "ccsid" option is "job", it
// is valid to specify it explicitly.
XML-SAX %handler(hdlr:comm) %XML(xmlString : 'ccsid=job');

// In the following example, the data is to be
// returned in UCS-2.
opt = 'ccsid=ucs2';
XML-SAX %handler(hdlr:comm) %XML(xmldata : opt);

// In the following example, the data is to be
// returned in UTF-8.  The handling procedure must
// exercise caution to convert the data to some CCSID
// that the program can handle, if the data is to be
// used within the handling procedure.
XML-SAX %handler(hdlr:comm) %XML(xmldata : 'ccsid=1208');
```

Figure 418. Example of the ccsid option:

Note: For *XML_UCS2_REF and *XML_ATTR_UCS2_REF events, the data is always returned as a UCS-2 value independent of the *ccsid* option.

XML-SAX event-handling procedure

The event-handling procedure is a user-written prototyped procedure. It must have the following return type and parameters:

Parameter number or return value	Data type and passing mode	Description
Return value	4-byte integer (10I 0)	Returning a value of zero indicates that parsing should continue; returning any other value indictes that parsing should end.
1 – Communication area	Any type, passed by reference	Used to communicate between the XML-SAX operation and the handler, and between successive calls to the handler.
2 – Event	4-byte integer (10I 0), passed by value	The XML event discovered by the parser. Special words such as *XML_START_ELEMENT can be used to identify the events within the handling procedure. See "XML events" on page 889.
3 – Data	Pointer (*), passed by value	If this parameter is not relevant to the event, it will have a value of *NULL. Otherwise, it will point to the data for the event. For the *XML_UCS2_REF, and *XML_ATTR_UCS2_REF events, the data will always be UCS-2 data. For all other events, the data will be in the CCSID specified by the "ccsid" option of the %XML built-in function.

Parameter number or return value	Data type and passing mode	Description
4 – Length	8-byte integer (20I 0), passed by value	For most events, this is the length of the data pointed to by the third parameter, in bytes. If this parameter is not relevant for a particular event, it will have the value -1. If the data is being returned in UCS-2 due to the "ccsid" option of the %XML built-in function, this value must be divided by two to obtain the number of UCS-2 characters. For the *XML_EXCEPTION event, this parameter will have the length of the document that was parsed when the error occurred.
5 – Exception ID	4-byte integer (10I 0), passed by value	The exception ID. For all events other than *XML_EXCEPTION, this parameter will have a value of zero. See the section on XML return codes in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

See "%HANDLER (handlingProcedure : communicationArea)" on page 539 for more information on %HANDLER.

```
D saxHandler      pr            10i 0
D   commArea                          likeds(myCommArea)
D   event                       10i 0 value
D   string                        *   value
D   stringlen                   20i 0 value
D   exceptionId                 10i 0 value
```

Figure 419. Sample prototype for an XML-SAX handling procedure

XML events

During the SAX parse of your XML document, several XML events will be passed to your XML-SAX handling procedure. To identify the events within your procedure, use the special names starting with *XML, for example *XML_START_ELEMENT.

For most events, the handling procedure will be passed a value associated with the event. For example, for the *XML_START_ELEMENT event, the value is the name of the XML element.

Table 85. XML events

Event	Value
1. Events discovered before the first XML element	
*XML_START_DOCUMENT	Indicates that parsing has begun
*XML_VERSION_INFO	The "version" value from the XML declaration
*XML_ENCODING_DECL	The "encoding" value from the XML declaration
*XML_STANDALONE_DECL	The "standalone" value from the XML declaration
*XML_DOCTYPE_DECL	The value of the Document Type Declaration
2. Events related to XML elements	
*XML_START_ELEMENT	The name of the XML element that is starting
*XML_CHARS	The value of the XML element
*XML_PREDEF_REF	The value of a predefined reference
*XML_UCS2_REF	The value of a UCS-2 reference
*XML_UNKNOWN_REF	The name of an unknown entity reference
*XML_END_ELEMENT	The name of the XML element that is ending
3. Events related to XML attributes	
*XML_ATTR_NAME	The name of the attribute
*XML_ATTR_CHARS	The value of the attribute
*XML_ATTR_PREDEF_REF	The value of a predefined reference
*XML_ATTR_UCS2_REF	The value of a UCS-2 reference
*XML_UNKNOWN_ATTR_REF	The name of an unknown entity reference
*XML_END_ATTR	Indicates the end of the attribute
4. Events related to XML processing instructions	
*XML_PI_TARGET	The name of the target
*XML_PI_DATA	The value of the data
5. Events related to XML CDATA sections	
*XML_START_CDATA	The beginning of the CDATA section
*XML_CHARS	The value of the CDATA section
*XML_END_CDATA	The end of the CDATA section
6. Other events	
*XML_COMMENT	The value of the XML comment
*XML_EXCEPTION	Indicates that the parser discovered an error
*XML_END_DOCUMENT	Indicates that parsing has ended

This sample XML document is referred to in the descriptions of the XML events.

```
<?xml version="1.0" encoding="ibm-1140" standalone="yes" ?>
<!DOCTYPE page [
  <!ENTITY abc "ABC Inc">
]>
<!-- This document is just an example  -->
<sandwich>
  <bread type="baker's best" supplier="&abc;" />
  <?spread   please use real mayonnaise ?>
  <spices attr="&#x2B;">Salt &amp; pepper</spices>
  <filling>Cheese, lettuce,
         tomato, &#0061; &xyz;
  </filling>
  <![CDATA[We should add a <relish> element in future!]]>
</sandwich>junk
```

Figure 420. Sample XML document referred to in the descriptions of the XML events

***XML_START_DOCUMENT**
This event occurs once, at the beginning of parsing the document. Only the first two parameters are relevant for this event. Accessing the String parameter will cause a pointer-not-set error to occur.

***XML_VERSION_INFO**
This event occurs if the XML declaration contains version information. The value of the string parameter is the version value from the XML declaration.

From the example:
'1.0'

***XML_ENCODING_DECL**
This event occurs if the XML declaration contains encoding information. The value of the string parameter is the encoding value from the XML declaration.

From the example:
'ibm-1140'

***XML_STANDALONE_DECL**
This event occurs if the XML declaration contains standalone information. The value of the string parameter is the standalone value from the XML declaration.

From the example:
'yes'

***XML_DOCTYPE_DECL**
This event occurs if the XML declaration contains a DTD (Document Type Declaration). Document type declarations begin with the character sequence '<!DOCTYPE' and end with a '>' character.

Note: This is the only event where the XML text includes the delimiters.

The value of the string parameter is the entire DOCTYPE value, including the opening and closing character sequences.

From the example
`'<!DOCTYPE page [LF  <!ENTITY abc "ABC Inc">LF]>'`

(*LF* represents the LINE FEED character.)

***XML_START_ELEMENT**
This event occurs once for each element tag or empty element tag. The value of the string parameter is the element name.

From the example, in the order they appear:

1. 'sandwich'
2. 'bread'
3. 'spices'
4. 'filling'

***XML_CHARS**
This event occurs for each fragment of content. Content normally consists of a single string, even if the text is on multiple lines. It is split into multiple events if it contains references. The value of the string parameter is the fragment of the content.

From the example:

1. 'Salt '
2. ' pepper'
3. 'Cheese, lettuce,WWWtomato, ', where WWW represents several "whitespace" characters. See the “Notes” section.
4. 'We should add a <relish> element in future!'

Notes:

1. The content fragment '&' causes a *XML_PREDEF_REF event, and the fragment '=' causes a *XML_UCS2_REF event.
2. If the value spans multiple lines of the XML document, it will contain end-of-line characters and it will possibly contain unwanted series of blanks. In the example, "lettuce," and "tomato" are separated by a line-feed character and several blanks. These characters are called *whitespace*; whitespace is ignored if it appears between XML elements, but it is considered to be data if it appears within an element. If it is possible that the XML data may contain unwanted whitespace, the data may need to be trimmed before use. To trim unwanted leading and trailing whitespace, use the following coding. See example Figure 424 on page 901.

```
 * x'15'=newline  x'05'=tab     x'0D'=carriage-return
 * x'25'=linefeed x'40'=blank

D whitespaceChr   C                   x'15050D2540'
 /free
     temp = %trim(value : whitespaceChr);
```

***XML_PREDEF_REF**
This event occurs when content has one of the predefined single-character references '&', ''', '>', '<', and '"'. The value of the string parameter is the single-byte character:

Table 86.

&	&
'	'
>	<
<	>
"	"

Note: The string is a UCS-2 character if the parsing is being done in UCS-2.

From the example:
'&', from the content for the "spices" element.

***XML_UCS2_REF**
This event occurs when content has a reference of the form "" or "", where 'd' and 'h' represent decimal and hexadecimal digits, respectively. The value of the string parameter is the UCS-2 value of reference.

Note: This parameter is a UCS-2 character (type C) even if the parsing is being done in single-byte character.

From the example:
The UCS-2 value '=', appearing as "=", from the fragment at the end of the "filling" element,

***XML_UNKNOWN_REF**
This event occurs for an entity reference appearing in content, other than the five predefined entity references as shown for *XML_PREDEF_REF above. The value of the string parameter is the name of the reference; the data that appears between the opening '&' and the closing ';'.

From the example:
'xyz'

***XML_END_ELEMENT**
This event occurs when the parser finds an element end tag or the closing angle bracket of an empty element. The value of the string parameter is the element name.

From the example, in the order they occur:

1. 'bread'
2. 'spices'
3. 'filling'
4. 'sandwich'

***XML_ATTR_NAME**
This event occurs once for each attribute in an element tag or empty element tag, after recognizing a valid name. The value of the string parameter is the attribute name.

From the example, in the order they appear:

1. 'type'
2. 'supplier'
3. 'attr'

***XML_ATTR_CHARS**
This event occurs for each fragment of an attribute value. An attribute value normally consists of a single string, even if the text is on multiple lines. It is split into multiple events if it contains references. The value of the string parameter is the fragment of the attribute value.

From the example, in the order they appear:

1. 'baker'
2. 's best'

Notes:

1. The fragment ''' causes a *XML_ATTR_PREDEF_REF event
2. See the discussion on “*XML_CHARS” on page 892 for recommendations for handling unwanted end-of-line characters and unwanted blanks.

***XML_ATTR_PREDEF_REF**

This event occurs when an attribute value has one of the predefined single-character references '&', ''', '>', '<', and '"'. The value of the string parameter is the single-byte character:

Table 87.

&	&
'	'
>	<
<	>
"	"

Note: The string is a UCS-2 character if the parsing is being done in UCS-2.

From the example, the value for the "type" attribute:

' (The apostrophe character, "&apos")

***XML_ATTR_UCS2_REF**

This event occurs when an attribute value has a reference of the form '&#dd..;' or '&#xhh..;', where 'd' and 'h' represent decimal and hexadecimal digits, respectively. The value of the string parameter is the UCS-2 value of the reference.

Note: This parameter is a UCS-2 character (type C) even if the parsing is being done in single-byte character.

From the example, from the value of the "attr" attribute:

The UCS-2 value '+', appearing as "+" in the document.

***XML_UNKNOWN_ATTR_REF**

This event occurs for an entity reference appearing in an attribute, other than the five predefined entity references as shown for *XML_ATTR_PREDEF_REF above. The value of the string parameter is the name of the reference; the data that appears between the opening '&' and the closing ';'.

From the example:

'abc'

Note: The parser does not parse the DOCTYPE declaration, so even though entity "abc" is defined in the DOCTYPE declaration, it is considered undefined by the parser.

***XML_END_ATTR**

This event occurs when the parser reaches the end of an attribute value. The string parameter is not relevant for this event. Accessing the string parameter will cause a pointer-not-set error to occur.

From the example:

For the attribute type="baker's best", the *XML_END_ATTR event occurs after all three parts of the attribute value ("baker", ' and "s best") have been handled.

***XML_PI_TARGET**

This event occurs when the parser recognizes the name following the processing instruction (PI) opening character sequence '<?'. Processing instructions allow XML documents to contain special instructions for applications. The value of the string parameter is the processing instruction name.

From the example:
'spread'

***XML_PI_DATA**

This event occurs for the data part of a processing instruction, up to but not including the PI closing character sequence '?>'. The value of the string parameter is the processing instruction data, including trailing but not leading white space.

From the example:
'please use real mayonnaise '

Note: See the discussion for "*XML_CHARS" on page 892 for recommendations for handling unwanted end-of-line characters and unwanted blanks.

***XML_START_CDATA**

This event occurs when a CDATA section begins. CDATA sections begin with the string '<![CDATA[' and end with the string ']]>'. Such sections are used to "escape" blocks of text containing characters that would otherwise be recognized as XML markup. The parser passes the content of a CDATA section between these delimiters as a single *XML_CHARS event. The value of the string parameter is always the opening character sequence '<![CDATA['.

From the example:
`'<![CDATA['`

***XML_END_CDATA**

This event occurs when a CDATA section ends. The value of the string parameter is always the closing character sequence ']]>'.

From the example:
']]>'

***XML_COMMENT**

This event occurs for any comments in the XML document. The value of the string parameter is the data between the opening delimiter '<!--' and the closing delimiter '-->' , including leading and trailing white space.

From the example:
' This document is just an example '

***XML_EXCEPTION**

This event occurs when the parser detects an error. The value of the string parameter is the "String" parameter is not relevant for this event. Accessing the String parameter will cause a pointer-not-set error to occur. The value of the string-length parameter is the length of the document that was parsed up to and including the point where the exception occurred. The value of the Exception-Id parameter is the exception ID as assigned by the parser. The meaning of these exceptions is documented in the section on XML return codes in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

From the example:
An exception event would occur when the parser encountered the word "junk", which is non-whitespace data appearing after the end of the XML document. (The XML document ends with the end-element tag for the "sandwich" element.)

***XML_END_DOCUMENT**
This event occurs when parsing has completed. Only the first two parameters are relevant for this event. Accessing the String parameter will cause a pointer-not-set error to occur.

Note: To aid in debugging an XML-SAX handling procedure, the Control specification keyword DEBUG(*XMLSAX) can be specified. For more details on this keyword, see "DEBUG{(*INPUT | *DUMP | *XMLSAX | *NO | *YES)}" on page 263 and the Debugging chapter in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*. For more information about XML parsing, including limitations of the XML parser used by RPG, see the XML chapter in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

Examples of the XML-SAX operation

```
D xmlString       S              C    '<?xml version="1.0"> +
D                                     <elem>data</elem>'
D psds            DS
D   xmlRc                      10I 0  OVERLAY(psds:368)
 /free
   // The XML is in an IFS file.  The "option" operand of %XML specifies
   // that the document operand is the name of an IFS file.
   XML-SAX %HANDLER(mySaxHandler : myHandlerInfo)
           %XML('/home/myuserid/myxml.xml' : 'doc=file');

   // The XML is in a string.  The "option" operand of %XML is not specified.
   XML-SAX %HANDLER(mySaxHandler : myHandlerInfo) %XML(xmlString);
```

Figure 421. XML-SAX operations in Free-form calculations

```
CL0N01Factor1+++++++Opcode&ExtExtended-Factor2+++++++++++++++++++++++++
C                   XML-SAX   %HANDLER(mySaxHandler : myHandlerInfo)
C                             %XML('/home/myuserid/myxml.xml' : 'doc=file')

C                   XML-SAX   %HANDLER(mySaxHandler : myHandlerInfo)
C                             %XML(xmlString)
```

Figure 422. XML-SAX operations in Fixed-form calculations

```
H DEBUG(*XMLSAX)
Fqsysprt   o    f  132        printer

 * The xmlRc subfield will be set to a non-zero value
 * if the XML-SAX operation fails because of an error
 * discovered by the parser

D psds          SDS
D   xmlRc
 [1]                     10I 0  OVERLAY(psds:368)

D qsysprtDs       DS           132

 * This data structure defines the type for the parameter
 * passed to the SAX handling procedure.
 [2]
D value_t         S             50A    VARYING
D handlerInfo_t   DS                   QUALIFIED
D                                      BASED(dummy)
D   pValue                       *
D   numAttendees                5P 0
D   name                               LIKE(value_t)
D   company                            LIKE(value_t)
D   alwExtraAttr                1N
D   handlingAttrs...
D                                N

 * Define a specific instance of the handlerInfo_t data
 * structure and the prototype for the handler
D myHandlerInfo   DS                   LIKEDS(handlerInfo_t)
D mySaxHandler    PR            10I 0
D   info                               LIKEDS(handlerInfo_t)
D   event                       10I 0  VALUE
D   stringPtr                     *    VALUE
D   stringLen                   20I 0  VALUE
D   exceptionId                 10I 0  VALUE


 /free
    monitor;
       // Start XML parsing
       // Indicate that the handler should not allow
       // any unexpected attributes in the XML elements.
       myHandlerInfo.alwExtraAttr = *OFF;

 [3]   XML-SAX %HANDLER(mySaxHandler : myHandlerInfo)
                %XML('/home/myuserid/myxml.xml' : 'doc=file');
       // The XML parse completed normally
       // Results are passed back in the communication
       // area specified by the %HANDLER built-in function
       qsysprtDs = 'There are '
                 + %CHAR(myHandlerInfo.numAttendees)
                 + ' attendees.';
    on-error 00351;
       // The XML parse failed with a parser error.
       // The return code from the parser is in the PSDS.
```

Figure 423. A complete working program, illustrating an XML-SAX handling procedure (Part 1 of 4)

```
         qsysprtDs = 'XML parser error: rc='
                   + %CHAR(xmlRc)
                   + '.';
      endmon;

      write qsysprt qsysprtDs;
      *inlr = '1';
  /end-free

 P mySaxHandler    B
 D                 PI            10I 0
 D   info                              LIKEDS(handlerInfo_t)
 D   event                       10I 0  VALUE
 D   stringPtr                     *    VALUE
 D   stringLen                   20I 0  VALUE
 D   exceptionId                 10I 0  VALUE

 D value           S                   LIKE(value_t)
 D                                     BASED(info.pValue)

 D chars           S          65535A   BASED(stringPtr)
 D ucs2            S          16383C   BASED(stringPtr)
 D ucs2Len         S             10I 0

  /free

   select;
```

Figure 423. A complete working program, illustrating an XML-SAX handling procedure (Part 2 of 4)

```
   // start parsing
   when event = *XML_START_DOCUMENT;
[4]
     clear info;

   // start processing an attendee, by indicating
   // that subsequent calls to this procedure should
   // handle XML-attribute events.
   when event = *XML_START_ELEMENT;
     if %subst(chars : 1 : stringLen) = 'attendee';
        info.handlingAttrs = *ON;
[5]
        info.name = '';
        info.company = '';
        info.numAttendees += 1;
     endif;

   // display information about the attendee
   when event = *XML_END_ELEMENT;
     if %subst(chars : 1 : stringLen) = 'attendee';
        info.handlingAttrs = *OFF;
        qsysprtDs = 'Attendee '
                  + info.name
                  + ' is from company '
                  + info.company;
        write qsysprt qsysprtDs;
     endif;

   // prepare to get an attribute value by setting
   // a basing pointer to the address of the correct
   // variable to receive the value
   when event = *XML_ATTR_NAME;
     if info.handlingAttrs;
       if %subst(chars : 1 : stringLen) = 'name';
          info.pValue = %addr(info.name);
       elseif %subst(chars : 1 : stringLen) = 'company';
          info.pValue = %addr(info.company);
       else;
          // If the XML element is not expected to have
          // extra attributes, halt the parsing by
          // returning -1.
          if not info.alwExtraAttr;
             qsysprtDs = 'Unexpected attribute '
                       + %subst(chars : 1 : stringLen)
                       + ' found.';
             write qsysprt qsysprtDs;
             return -1;
[6]
          endif;
          info.pValue = *NULL;
       endif;
    endif;
```

Figure 423. A complete working program, illustrating an XML-SAX handling procedure (Part 3 of 4)

```
    // handle an exception
    when event = *XML_EXCEPTION;
        qsysprtDs = 'Exception '
                  + %char(exceptionId)
                  + ' occurred.';
        write qsysprt qsysprtDs;
        return exceptionId;

    other;

      // If this is an attribute we are interested
      // in, the basing pointer for "value" has been
      // set to point to either "name" or "company"

      // Append each fragment of the value to the
      // current data

      if  info.handlingAttrs
      and info.pValue <> *NULL;
        if event = *XML_ATTR_CHARS
        or event = *XML_ATTR_PREDEF_REF;
           value += %subst(chars : 1 : stringLen);
        elseif event = *XML_ATTR_UCS2_REF;
           ucs2Len = stringLen / 2;

 [7]     value += %char(%subst(ucs2 : 1 : ucs2Len));
        endif;
      endif;
    endsl;

    return 0;
 [8]

 /end-free
P mySaxHandler    E
```

Figure 423. A complete working program, illustrating an XML-SAX handling procedure (Part 4 of 4)

This example illustrates several features of SAX parsing.

1. The "External Return Code" subfield of the PSDS, named xmlRc here.
2. The communication area data structure, used to communicate between the XML-SAX operation and the SAX event-handling procedure.
3. The XML-SAX operation initiates the parsing of the XML document.
4. The SAX event-handling procedure compares the event parameter to the special names *XML_START_DOCUMENT etc.
5. The communication area is also used for the event-handling procedure to communicate with itself between calls.
6. The event-handling procedure discovers an error and halts the parsing by returning -1.
7. The *XML_ATTR_UCS2_REF event has UCS-2 data, independent of the CCSID that is normally used to return data for this XML-SAX operation. The length represents the number of bytes in the data, so it must be divided by two to obtain the number of UCS-2 characters.
8. If the event-handling procedure does not discover any errors, it returns 0, indicating that parsing should continue.

The following sample XML document could be used with this example.

```
<meeting>
  <attendee name="Jack" company="A&amp;B Electronics"/>
  <attendee company="City&#x2B; Waterworks" name="Jill"/>
  <attendee name="Bill" company="Ace Movers" extra="yes"/>
</meeting>
```

```
 // The following procedure returns a string that is the same
 // as the input string except that strings of whitespace are
 // converted to a single blank.

P rmvWhiteSpace   b
D rmvWhiteSpace   pi         65535a   varying
D   input                    65535a   varying const
D output          s                   like(input) inz('')

 * x'15'=newline  x'05'=tab     x'0D'=carriage-return
 * x'25'=linefeed x'40'=blank
D whitespaceChr   C                   x'15050D2540'
D c               s              1A
D i               s             10I 0
D inWhitespace    s              N    INZ(*OFF)
 /free
      // copy all non-whitespace characters to the return value
      for i = 1 to %len(input);
         c = %subst(input : i : 1);
         if %scan(c : whitespaceChr) > 0;
            // If this is a new set of whitespace, add one blank
            if inWhitespace = *OFF;
               inWhitespace = *ON;
               output += ' ';
            endif;
         else;
            // Not handling whitespace now.  Add character to output
            inWhitespace = *OFF;
            output += c;
         endif;
      endfor;
      return output;
 /end-free
P rmvWhiteSpace   e
```

Figure 424. Removing internal whitespace from XML data

For more information about XML operations, see “XML Operations” on page 475.

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 1 is not used. Factor 2 must be numeric and can contain one of: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric, and can contain one of: an array, array element, field, subfield, or table name.

Half-adjust can be specified.

For the rules for the Z-ADD operation, see “Arithmetic Operations” on page 434.

See Figure 172 on page 437 for an example 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. You can use the operation to change the sign of a field. Factor 1 is not used. Factor 2 must be numeric and can contain one of the following: an array, array element, field, figurative constant, literal, named constant, subfield, or table name.

The result field must be numeric, and can contain one of the following: an array, array element, field, subfield, or table name.

Half-adjust can be specified.

For the rules for the Z-SUB operation, see "Arithmetic Operations" on page 434.

See Figure 172 on page 437 for an example of the Z-SUB operation.

Part 5. Appendixes

Appendix A. RPG IV Restrictions

Function	Restriction
Array/table input record length for compile time	Maximum length is 100
Character field length	The maximum length for a fixed-length character field is 16773104. The maximum length for a variable-length character field is 16773100.
Graphic or UCS-2 field length	The maximum length for a fixed-length graphic or UCS-2 field is 8386552. The maximum length for a variable-length graphic or UCS-2 field is 8386550.
Control fields (position 63 and 64 of input specifications) length	Maximum length is 256
Named data structure length	Maximum of 16773104
Unnamed data structure length	Maximum of 16773104
Data structure occurrences (number of)	Maximum of 16773104 per data structure; the maximum total size is 16773104.
Edit Word	Maximum length of 115
Elements in an array/table (DIM keyword on the definition specifications)	Maximum of 16773104 per array; the maximum total size is 16773104.
Levels of nesting in structured groups	Maximum of 100
Levels of nesting in expressions	Maximum of 100
Look-ahead	Can be specified only once for a file. Can be specified only for primary and secondary files.
Named Constant or Literal	Maximum length of 16380 characters for a character or hexadecimal literal, 16379 DBCS characters for a graphic literal, 8190 UCS-2 characters for a UCS-2 literal, and 63 digits with 63 decimal positions for a numeric literal.
Overflow indicator	Only 1 unique overflow indicator can be specified per printer file.
Parameters to programs	Maximum of 255
Parameters to procedures	Maximum of 399
Primary file (P in position 18 of file description specifications)	Maximum of 1 per program
Printer file (PRINTER in positions 36 through 42 of file description specifications)	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 address file (R in position 18 of file description specifications)	Only 1 allowed per program.
Record length for a file	Maximum length is 99999 [1]
Structured groups (see levels of nesting)	

Function	Restriction
Storage allocation	Maximum length is 16776704 [2]
Symbolic names	Maximum length is 4096

Notes:

1. Any device record size restraints override this value.
2. The practical maximum is normally much less.

Appendix B. EBCDIC Collating Sequence

Table 88. EBCDIC Collating Sequence

Ordinal Number	Symbol	Meaning	Decimal Represen-tation	Hex Represen-tation
65	ƀ	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	|	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	ª	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

Table 88. EBCDIC Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Represen-tation	Hex Represen-tation
126	'	Apostrophe, prime sign	125	7D
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

Table 88. EBCDIC Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Represen- tation	Hex Represen- tation
. . .				
193	{	Left brace	192	C0
194	A		193	C1
195	B		194	C2
196	C		195	C3
197	D		196	C4
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

Table 88. EBCDIC Collating Sequence (continued)

Ordinal Number	Symbol	Meaning	Decimal Represen-tation	Hex Represen-tation
. . .				
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
Note: These symbols may not be the same for all codepages. Codepages may map different hexadecimal values to different symbols for various languages. For more information, see the iSeries Information Center globalization topic.				

Bibliography

For additional information about topics related to ILE RPG programming, refer to the following publications:

- *CL Programming*, SC41-5721-06, provides a wide-ranging discussion of programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.

  See the iSeries Information Center programming category (URL http://www.ibm.com/systems/i/infocenter/) for a description of the IBM icontrol language (CL) and its commands.
- *Communications Management*, SC41-5406-02, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.

  See the iSeries Information Center database and file systems category for related database programming topics such as, using files in application programs, database organization, data description specifications (DDS) and DDS keywords, distributed data management (DDM), and application programming interfaces.
- *Experience RPG IV Multimedia Tutorial*, GK2T-9882-00 is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG code examples are shipped with the tutorial and run directly on the operating system.
- *ILE Concepts*, SC41-5606-09, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*, SC09-2507-08, provides information about the ILE RPG programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE). It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More*, SG24-5402 provides hints and tips for system programmers who want to take full advantage of RPG IV and the Integrated Language Environment (ILE).

You can obtain current IBM i information and publications from the **IBM i Information Center** at the following Web site:

`http://www.ibm.com/systems/i/infocenter/`

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 described in this document. The furnishing of this document does not grant 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.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

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:

IBM Canada Ltd. Laboratory
Information Development
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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, 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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This publication is intended to help you create programs using RPG IV source. This publication documents General-Use Programming Interface and Associated Guidance Information provided by the ILE RPG compiler.

General-Use programming interfaces allow the customer to write programs that obtain the services of the ILE RPG compiler.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

Q

R

S

V

W

X

Y

Z

Product Number: 5770-WDS

Printed in U.S.A.

SC09-2508-08