precisely

EngageOne Enrichment

Developer Guide

Version 7.4.1

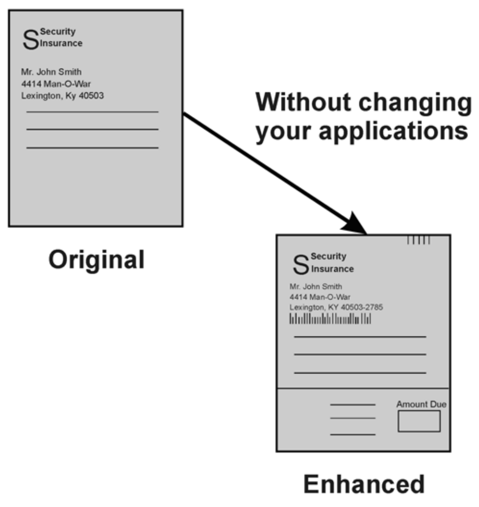
Table of Contents

1 - Introducing Enrichment

In this section

What is Enrichment

Enrichment processes and manipulates print streams and data to support advanced printing and distribution strategies. Enrichment modifies the output from existing applications independently; therefore, it requires no changes to your business applications.



Enrichment significantly increases productivity and reduces turnaround to implement new print processing applications.

Enrichment can handle a variety of output types from your business applications: flat files, AFP line-data files, AFP Mixed Mode files, fully composed AFPDS files, Xerox DJDE, Metacode files, PCL files, PDF, PostScript and so on. Because only the output from your business applications is processed, you do not need to change the business applications themselves.

What Can I Do with Enrichment

Enrichment allows you to perform many print stream enhancement functions, including the following:

- Content enhancement
  - Add personalized messages
  - Move text and change fonts
  - Add new text in white space
  - Move and modify barcodes
  - Add images and graphics
- Postal automation

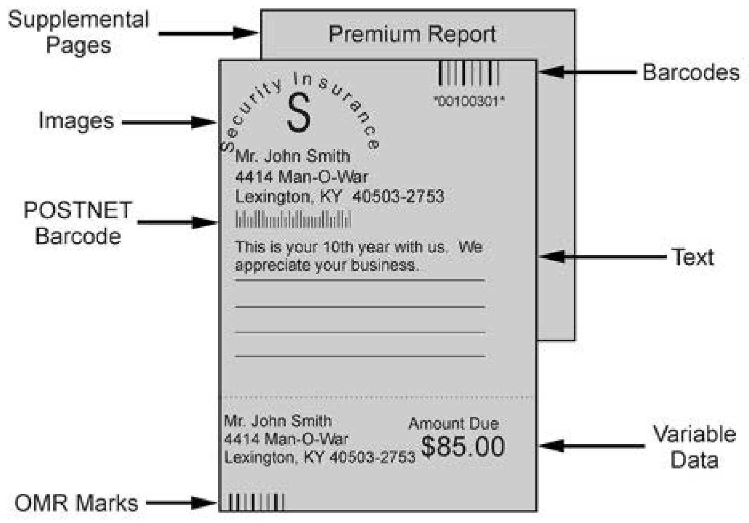
- Add POSTNET™ barcodes
- Add 4-State and Intelligent Mail® barcodes
- Add ChinaPost barcodes
- Add ZIP + 4®
- Standardize addresses, when used in conjunction with an address validation product such as Finalist
- Postal presorting, when used in conjunction with a postal presort product such as Mailstream Plus
- Move update processing
- Combine mail streams produced from multiple applications into a single mail stream

- Inserter control
  - Remove or modify old barcodes
  - Add advanced barcodes
  - Add sequence numbers
  - Build file-based insertion controls

- Sort print streams
- Consolidate/merge print streams

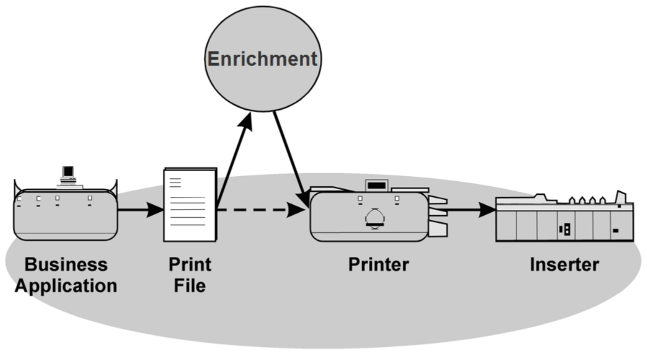
For sample applications that show how to accomplish some of these tasks, see the *Enrichment Sample Applications Guide*.

The following illustration demonstrates several kinds of enhancements you can make to a print stream using Enrichment.



How Does Enrichment Work

Enrichment typically just plugs into your existing process. This usually means adding an extra step to the JCL, UNIX shell script, or Windows batch file you use for print processing. You can also run Enrichment as a totally independent application.



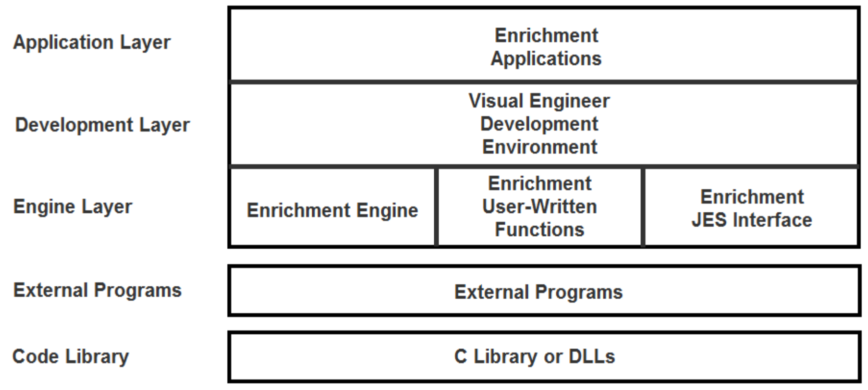
Without Enrichment, your business application creates a print file that prints and runs through an inserter. If changes to the content or appearance of the print file are required, you must go back to the business application and start over. This can be costly and time-consuming.

With Enrichment, your business application creates a print file that you run through Enrichment for the appropriate enhancements. The enhanced output prints and runs through the inserter.

Note: Enrichment does not transform print streams from one format to another (for instance, from AFP to Metacode).

Enrichment Architecture

The following diagram illustrates the Enrichment architecture.



C Library or DLLs

The C library (for mainframe systems or UNIX) or DLLs (for Windows) serve as the code library accessed by the Enrichment engine when running applications. Depending on the platform, various libraries may be required.

External Programs

External programs include presorting, sorting and mail cleansing software called by Enrichment. These programs are not considered part of Enrichment.

Enrichment Engine

The Enrichment engine is the component that processes print streams according to instructions coded in the Enrichment language (control file and rules). The engine runs on a variety of platforms. Code created to run on the engine under one platform will run on all others unless it accesses platform-dependent data, external programs or user-written functions. This book describes coding methods for the Enrichment engine.

Enrichment User-Written Functions

Users can extend Enrichment functionality by writing their own functions in a variety of languages. The object code for the user-written functions must be executable code that has been compiled and linked. On Windows, it may be a DLL.

Enrichment JES Interface

The JES Interface provides the ability for JES spool volume data to be used as input to an Enrichment application. There are two JES interfaces: a non-SAPI interface and a SAPI interface. The non-SAPI interface directly accesses the JES spool volumes. The SAPI interface uses the IBM SYSOUT Application Programming Interface (SAPI) to retrieve spool data, and does not directly access spool volumes.

Enrichment Visual Engineer Development Environment

The Enrichment Visual Engineer development environment is a tool used by developers to build and test Enrichment applications. It runs on Windows and can improve development productivity compared to manually coding and testing applications in a standard editor.

Enrichment Applications

Programmers or printer specialists write Enrichment applications to perform specific functions on specific print files. Each application is coded using the Enrichment language, which includes two basic components:

- **Control File:** defines the objects to be processed by Enrichment using object-oriented constructs.
- **Rule File:** defines conditional processing logic for the application in traditional programming code. While a rule file is optional, it is used for most applications. The rule file is composed of sections executed at different points in the application. In many cases the rule file is embedded in the control file so it is not an actual file. However, the rule file can be a separate file that is called from a control file.

Some users build programs that automatically write Enrichment code and generate Enrichment applications.

Print Streams

Enrichment supports the following print stream formats:

- AFP
- DJDE
- Fixed length record line data
- Line Data
- Metacode
- PCL
- PDF
- PostScript

AFP Print Streams

AFP is a printer control language for AFP-compatible printers. There are several different forms of AFP:

- **AFP mixed mode:** a mix of line printer data and AFP records
- **AFP line data**: a data file with an associated map (PAGEDEF) that tells the printer where to put each data field on the page AFPDS A series of print commands in compact form

An AFP record contains one or more AFP commands. There are many different types of AFP commands. Each command has a specified layout and associated data. You can use Enrichment to modify the contents of these commands or add new AFP records. Modification of commands is normally done with a field replacement process. New records are automatically added when you use the Add group tags. However, you can format records and add them yourself using a variety of methods.

> **Note:** You can use Enrichment Visual Engineer's Explain function to view and get more information on the AFP commands in a print stream.

Given that AFP is an IBM printer control language and is normally associated with mainframe systems, AFP files are normally in EBCDIC format.

Refer to your IBM documentation for more information on AFP commands.

DJDE Print Streams

Xerox DJDE data is a special record which is placed within existing line data printer files to add special Xerox print features. These records contain Dynamic Job Descriptor Entries (DJDEs). DJDEs simply instruct the printer to perform a function. These commands are not printed.

Records containing DJDEs are identified to the printer by specifying a special character sequence in a specific range of columns. In the following example, the characters #+#+DJDE are used in columns 3 through 10 to indicate a DJDE record.

```
  #+#+DJDE ASSIGN=(1,3);
  #+#+DJDE ASSIGN=(2,11);
  #+#+DJDE ASSIGN=(6,23);
  #+#+DJDE ASSIGN=(7,73);
  #+#+DJDE ASSIGN=(8,80);
  #+#+DJDE BEGIN=(9.0,0.4);
  #+#+DJDE BEGIN=(.3,0);
  #+#+DJDE FORMAT=P0635;
  #+#+DJDE MARGIN=(.05,IN);
  #+#+DJDE DUPLEX=NO,FORMS=(A064),END;
13             HEALTH INSURANCE, INC.
 3             444 CORPORATE DR
 3             BATAVIA, IL  60510
03             Telephone No. 123-223-5800
```

As shown above, DJDEs are used to specify the format of the page, margins, and duplex information. Some of the key DJDE commands are described below.

Refer to the *Xerox Production Print Mode PDL/DJDE Reference* for your printer for more information on DJDE commands.

Line Data

Line data print streams are print streams that are prepared for printing on line printers. Line data can contain carriage control characters and table reference characters (TRC) for spacing and font selections.

Fixed-Length

A fixed-length print stream is a line data print stream with no end-of-record indicator. Records are defined as a specific number of bytes rather than some other inherent record structure such as in AFPDS, or using an end of line indicator such as Line Data.

Metacode Print Streams

The Xerox printer control language Metacode allows users to place commands in their print stream that give printing instructions. Metacode commands control many things, including positioning of data on a page, orientation of data on the page, and font selection. Metacode print files can be very large, so Metacode commands and command options are represented in hexadecimal codes. Refer to your Xerox documentation for more information on Metacode printing.

Note: Metacode print streams can also contain Xerox DJDE commands. You should process Xerox print streams without Metacode commands as DJDE line data.

PCL

Printer Command Language (PCL) is a Hewlett-Packard language.

PDF

PDF is a page description language supported by Adobe Systems. For complete information, see **https://acrobat.adobe.com/us/en/acrobat/about-adobe-pdf.html**.

.

PostScript

PostScript is a page description language supported by Adobe Systems. For complete information, see **http://www.adobe.com/products/postscript/main.html**.

Carriage Controls

A carriage control character is an instruction to the printer that determines how many lines to skip before printing the next line. Initially, all printers used a common set of carriage control commands to dictate how they would operate. This common set was established by ANSI, and is called ANSI carriage controls.

IBM created a second set of controls to provide expanded print capabilities called "machine codes" or "machine carriage controls". Machine carriage controls are hexadecimal codes which, like ANSI carriage controls, occur in column 1 of the print stream.

Over time, more sophisticated printers added the ability to pre-program specific line positions to be associated with specific carriage control values. The printer could move to these line positions when a specific carriage control command occurred. The programmer could predefine positions on pre-printed forms and tell the printer to move to them without trying to count how many lines the printer had printed and thus how many it had to move. These controls are called channel controls and generally have the values 1 through 9 and A through C. IBM uses a file called a Forms Control Buffer (FCB) to associate positions with each of the channel control values.

The following table lists valid ANSI carriage control characters.

Table 1: ANSI Carriage Control Characters

ANSI	Action
+	Print without spacing (overstrike).
Blank	Space one line and print (single spacing).
0	Space two lines and print (double spacing).
-	Space three lines and print (triple spacing).
1	Skip to the first line on a new page and print.

ANSI	Action
2	Skip to the line position defined as Channel 2 and print.
3	Skip to the line position defined as Channel 3 and print.
4	Skip to the line position defined as Channel 4 and print.
5	Skip to the line position defined as Channel 5 and print.
6	Skip to the line position defined as Channel 6 and print.
7	Skip to the line position defined as Channel 7 and print.
8	Skip to the line position defined as Channel 8 and print.
9	Skip to the line position defined as Channel 9 and print.
A	Skip to the line position defined as Channel 10 and print.
B	Skip to the line position defined as Channel 11 and print.
C	Skip to the line position defined as Channel 12 and print.

The following table lists valid machine carriage control characters.

Table 2: Machine Carriage Control Characters

Machine	Action
01	Print without spacing (overstrike).
09	Print and space one line (single spacing).
11	Print and space two lines (double spacing).
19	Print and space three lines (triple spacing).
89	Print, then skip to the first line on a new page.
91	Print, then skip to the line position defined as Channel 2.

Machine	Action
99	Print, then skip to the line position defined as Channel 3.
A1	Print, then skip to the line position defined as Channel 4.
A9	Print, then skip to the line position defined as Channel 5.
B1	Print, then skip to the line position defined as Channel 6.
B9	Print, then skip to the line position defined as Channel 7.
C1	Print, then skip to the line position defined as Channel 8.
C9	Print, then skip to the line position defined as Channel 9.
D1	Print, then skip to the line position defined as Channel 10.
D9	Print, then skip to the line position defined as Channel 11.
E1	Print, then skip to the line position defined as Channel 12.
0B	Space one line without printing.
13	Space two lines without printing.
1B	Space three lines without printing.
8B	Skip to the first line on a new page without printing.
93	Skip to the line position defined as Channel 2 without printing.
9B	Skip to the line position defined as Channel 3 without printing.
A3	Skip to the line position defined as Channel 4 without printing.
AB	Skip to the line position defined as Channel 5 without printing.
B3	Skip to the line position defined as Channel 6 without printing.
BB	Skip to the line position defined as Channel 7 without printing.

Machine	Action
C3	Skip to the line position defined as Channel 8 without printing.
CB	Skip to the line position defined as Channel 9 without printing.
D3	Skip to the line position defined as Channel 10 without printing.
DB	Skip to the line position defined as Channel 11 without printing.
E3	Skip to the line position defined as Channel 12 without printing.
03	No operation.

So, if the print stream were impact data with plus signs in column 1 of the data, we would specify our Input group <TYPE> tag as follows:

```
<TYPE> IMPACT ANSI
```

Table Reference Characters (TRCs)

Some line data printers can handle multiple fonts within an input. On such printers, you can define a specific font for each line of print by specifying a font number for the line. This font information often goes in column two of the print stream (with column one containing carriage control information). The font information in column two is called the Table Reference Character (TRC) or the font index. The actual font associated with the TRC is specified in JCL or by other means.

By changing the TRC, you can change the font used to print the line. Likewise, when you add information, you may need to know which font to use.

2 - Enrichment Language Basics

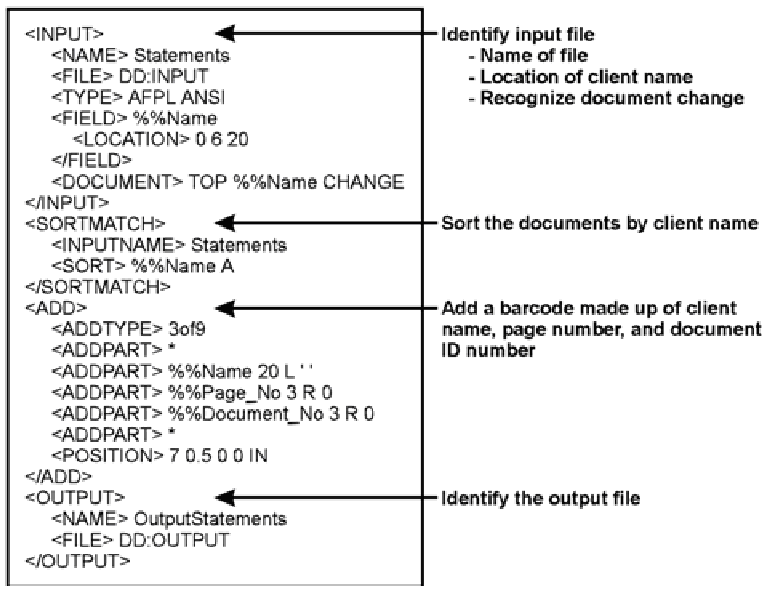
In this section

Enrichment Language Overview

Enrichment applications are written in a high-level object-based coding language.

Defining an Object

To define an object in Enrichment you use a tag group. The example below shows code from a control file. In the example, an input print stream object is defined using an Input tag group (`<INPUT>` and `</INPUT>`), and a data item object from the print stream is defined using a Field tag group (`<FIELD>` and `</FIELD>`) that is nested in the Input tag group. When you code a tag group, you are telling Enrichment to create and process the associated object.



```
<INPUT>
   <NAME> Statements
   <FILE> DD:INPUT
   <TYPE> AFPL ANSI
   <FIELD> %%Name
      <LOCATION> 0 6 20
   </FIELD>
   <DOCUMENT> TOP %%Name CHANGE
</INPUT>
<SORTMATCH>
   <INPUTNAME> Statements
   <SORT> %%Name A
</SORTMATCH>
<ADD>
   <ADDTYPE> 3of9
   <ADDPART> *
   <ADDPART> %%Name 20 L ' '
   <ADDPART> %%Page_No 3 R 0
   <ADDPART> %%Document_No 3 R 0
   <ADDPART> *
   <POSITION> 7 0.5 0 0 IN
</ADD>
<OUTPUT>
   <NAME> OutputStatements
   <FILE> DD:OUTPUT
</OUTPUT>
```

Identify input file
- Name of file
- Location of client name
- Recognize document change
Sort the documents by client name
Add a barcode made up of client name, page number, and document ID number
Identify the output file

Defining an Object's Attributes

If you were describing an object in nature—a tree, for example—you would need to mention certain attributes of that object (such as its type, size, age, color, and so forth). In Enrichment, you also need to describe each of the attributes of the object. To do this, you use tags within the object's tag group. These tags define specific attributes (such as the `<TYPE>` tag, which describes the type of input). Therefore, tags (and not tag groups) define data about the object.

Objects may also contain or be made up of other objects (such as the organization of a tree into branches). In Enrichment, you include tag groups within other tag groups to indicate hierarchical relationships. Because the Field group is within the Input group, this indicates that the field object described by a specific Field group is located only in the print stream associated with the Input group, as shown above.

Enrichment Language vs. Procedural Languages

The Enrichment language is unlike traditional procedural languages like COBOL or RPG. While the coding of an Enrichment object is similar to a declaration in a procedural language, in Enrichment the object is automatically processed when it is defined. Conversely, in a procedural language you would have to call functions or write code to read the input and find the data items on each page, because the declaration itself does not cause an action to occur. By design, Enrichment reads the print stream, finds pages, finds documents, and collects data from the document as a direct response to the definition of the `<INPUT>` object in the control file. This can save a lot of programming time and can result in much more reusable and maintainable code.

Note that the code you create in a rule file is procedural. However, these procedures are in effect acting upon the objects and processes established in the control file. For example, the `DOCUMENT:` section of the rule file operates on each of the documents read from the print stream. By simply redefining the value of a variable in the rule file, you can change the appearance of the document. You do not have to actually write code to place the variable on the document.

Tags and Tag Groups

The main component of the Enrichment language is the tag. Tags are used to identify input and output; to modify, delete and add objects; to specify sort and match criteria; to insert pages or records; and to set up address standardization and postal presort functions.

Tags and tag groups are keywords delimited by "<" and ">". For example, this is the file tag:

```
<FILE>
```

A tag is followed by one or more parameters. Multiple parameters are separated by spaces. Some parameters are required and others are optional, depending on the tag used.

Tag groups are sets of tags that go together to perform a particular function. There are two kinds of tag groups: major tag groups and minor tag groups. Major tag groups can be placed anywhere within the control file. Minor tag groups must be placed within (that is, nested) in another tag group.

Tag groups are defined by an opening and closing tag. The closing tag contains a "/" before the keyword. In the example below, the Banner tag group is used. The tags `<NAME>`, `<FILE>`, `<TYPE>`, and `<SUBSTITUTE>` are nested in the Banner tag group.

```
<banner>                        /* opening tag of Banner tag group */
<name> BANBEG                   /* Name tag followed by parameter */
<file> c:\banners\mybanner.lin  /* File tag followed by parameter */
<type> I                        /* Type tag followed by parameter */
<substitute> Y               /* Substitute tag followed by parameter */
</banner>                       /* closing tag of Banner tag group */
```

Note: Tags and tag groups are not case sensitive.

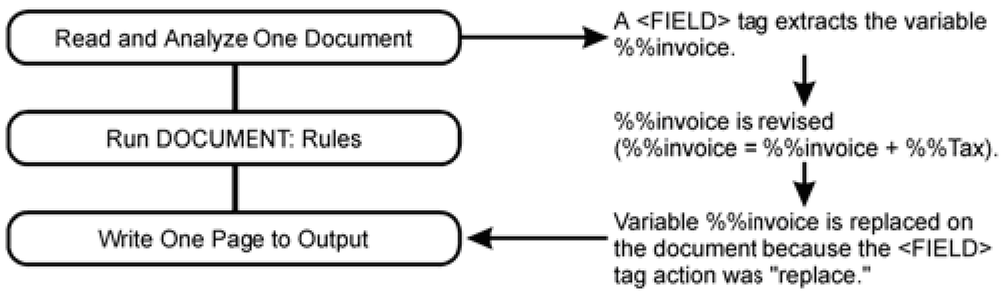
You can place tag groups in the control file in any order, but each tag group must begin and end with the proper tags (for example, `<INPUT>` and `</INPUT>` begin and end an Input tag group).

Variables

Variable names begin with %% and may contain up to 50 characters but no blanks. They are also case sensitive. The following are examples of different variable names.

```
%%InvoiceAmount
%%INVOICEAMOUNT
%%Invoice_Amount
```

All variables are available to all sections of the code. As the diagram below illustrates, if a `<FIELD>` tag in the control file sets a variable, its value is available to all subsequent steps in the application.


Read and Analyze One Document
Run DOCUMENT: Rules
Write One Page to Output
A <FIELD> tag extracts the variable %%invoice.
%%invoice is revised (%%invoice = %%invoice + %%Tax).
Variable %%invoice is replaced on the document because the <FIELD> tag action was "replace."

Variables are persistent. Once a variable's value is set, it retains that value until it is reset or the program ends.

For example, assuming *%%LAST_BRANCH* is not being assigned a value from the document by a `<FIELD>` tag but *%%BRANCH* is, the rule code shown in the following example would place a banner page between documents where the branch changes. Field variables are reset for each document.

```
START:
   %%LAST_BRANCH = ''
DOCUMENT:
   IF %%LAST_BRANCH <> %%BRANCH THEN
       <BANNER> BannerFile BEFORE
       %%LAST_BRANCH = %%BRANCH
   ENDIF
```

Note: In the above example, `%%LAST_BRANCH` retains its value from the last time the `DOCUMENT:` section was run.

You can place variable names within the body of inserts, documents, and banners. If there are variables within these objects, Enrichment can automatically replace the variable name with its value during processing.

For example, if you have `$%%amount` in the input document and `%%amount=10.56` then the text in your output would be `$10.56`. Likewise if you have `%%productname`™ in the input document and `%%productname=SuperWidget` then the text in your output would be `SuperWidget`™.

Note: Variable substitution is controlled by the `<SUBSTITUTE>` tag.

Note also that:

- Variables can contain numeric, string, array, or file name values. There is no typing of variables in Enrichment. Enrichment automatically detects a variable's expected type (integer, string, and so on) based on the context in which you use it.
- You do not need to declare a variable unless the variable is an array or a global variable. To declare an array or global variable, use the `DECLARE` function. For more information on arrays, see **Arrays** on page 21.
- To set a variable to nothing, enter two quotation marks with nothing between them. For example, `%%LAST_BRANCH = ''`.
- Delimit variable values that contain spaces with single quotes or double quotes—for example, `'908 West 7th St.'` or `"908 West 7th St."` If the value contains quotation marks or apostrophes, you must double the internal quotation marks (`'Guy''s Auto Hut'` or `"Bill''s ""Hi-Fi"" Stereos"`) or Enrichment assumes they terminate the value.

Initializing Variables in the Rules

If a variable is not set by a field and you're going to compare its value to that of another variable to set a condition, you may want to set that variable to a starting or initial value. This is called "initialization".

For example, assume we're using a counter (*%%OnePageDocs*) to keep track of the number of one-page documents in an output. We want the counter set initially to 0 so we can increment it every time Enrichment encounters a one-page document. As the following figure shows, we set *%%OnePageDocs* to 0 in the `START:` section of the rules and increment it in the `DOCUMENT:` section. We set *%%OnePageDocs* to a new variable (*%%Total1Pagers*) in the `FINISH:` section of the rules, perhaps to add this information to a report.

```
START:
   %%OnePageDocs = 0
DOCUMENT:
   IF %%TOTAL_PAGES = 1 THEN
       %%OnePageDocs = %%OnePageDocs + 1
   ENDIF
FINISH:
   %%Total1Pagers = %%OnePageDocs
```

System variables

Enrichment automatically creates variables called system variables. System variables are produced by key Enrichment steps. If a particular step, such as Presort, is not performed, then system variables normally produced by that step will have no value.

You can use system variables just as you would variables you create. System variables can be applied in your control or rule file. Control files and rule files share variables, so when you set a variable to a value in the control file, that variable and its value are available in the rule file.

Note: The value of system variables can change frequently. If a value is assigned to a system variable and later in the code you are expecting that value to still be in the system variable, it may not be because subsequent function calls or section changes could have modified the value.

Common System Variables

System variables are variables whose names have been reserved for use by Enrichment. Most often, their values are the result of internal calculations made during processing.

Enrichment automatically sets system variable values during processing, and their values can be set or changed in the rule file. Therefore, the type of information their values can contain is more restricted than for user-defined variables.

There are more than 80 system variables available for your Enrichment applications, but as the following table shows, the list of those most commonly used is quite small.

Table 3: Frequently-Used System Variables

System Variable	Value
%%DOCUMENT_NO	The overall document number, regardless of which output contains the document.
%%PAGE_NO	The logical front page number of the current page in the document.
%%RC	The return code from rule function call processing.
%%SEQUENCE_NO	The sequence number for each document in the output file.
%%TOTAL_PAGES	The total number of logical front pages in the document.

Arrays

To declare an array, use the `DECLARE` function. For complete information on the `DECLARE` function, see the*Enrichment Language Reference Guide*.

The elements of an array are referenced as *varname[index]*. For example, the elements of an array named *%%AR* are referenced as *%%AR[0]*, *%%AR[1]*, etc.

Array indexing is zero based. For example, if you declare an array with an array size of 5, valid indexes would be 0, 1, 2, 3, 4. Five would be out of bounds.

The following sample illustrates a basic array.

```
<input>
  <name> input
  <file> input.txt
  <type> I
</input>
<rule>
<content>
  %%result = declare(%%arr, 'A' , 5)
   %%arr[0] = "apple"
   %%arr[1] = "banana"
   %%arr[2] = "orange"
   %%arr[3] = "pineapple"
   %%arr[4] = "watermelon"
   FOR %%i = 0 to arraysize(%%arr) -1
      write("result.txt", %%arr[%%i])
   NEXT %%i
</content>
</rule>
```

```
<output>
  <name> output
  <file> output.txt
</output>
```

Functions

Enrichment includes an extensive set of functions that allow you to manipulate data from your inputs and return the result as a variable value. Enrichment also allows calls to user-written functions.

There are four types of functions: logical, string, numeric, and command.

- Logical functions return `TRUE (1)` or `FALSE (0)`.
- String functions return a string of characters.
- Numeric functions return a numeric value.
- Command functions are used to reference Enrichment functions that do not return a value. The following example shows the `WRITE` function which returns no value:

  ```
  WRITE(DD:EXCEPTIO, %%RECORD, F, 132)
  ```

  For compatibility with previous releases of EnrichmentEnrichment, function statements can also be referenced in a set statement. For example:

  ```
  %%X = WRITE(DD:EXCEPTIO, %%RECORD, F, 132)
  ```

  The variable on the left side is set to a null string.

The most commonly used functions include:

- `CHANGED` indicates whether a variable value has changed from the previous document to the current one.
- `FOUND` indicates whether a particular field was found in the document during Enrichment processing.
- `JUSTIFY` returns a string aligned within a specified number of characters.
- `LOOKUP` returns the record that satisfied the lookup criteria.
- `SUBSTR` returns a portion of a string.

Arguments

The data and options passed to a function are called arguments. Generally, arguments are numeric values, strings, or logical expressions. There are only two functions for which this is not the case: `CHANGED` and `FOUND`. These functions' arguments must be a variable because these functions supply information about the status of a variable.

The syntax for each Enrichment function is:

`FUNCTION(`*`arg1`*`,[`*`arg2`*`,`*`arg3`*`,`*`arg4`*`,`*`arg5`*`,`*`arg6`*`,`*`arg7`*`])`

In this example, `FUNCTION` has seven arguments. Arguments within brackets are optional. Enrichment uses the default value for optional arguments left unspecified.

> **Note:** Do not include brackets when specifying functions in a rule file.

Some arguments have default values that Enrichment uses if you do not explicitly specify the argument. For example, if a function call is made as follows:

```
%%answer = FUNCTION(arg1,,arg3)
```

Enrichment sets the value for *arg2* to its default. Note that no value was entered for *arg2*. Likewise, *arg2* and *arg3* could be set to their default values as follows:

```
%%answer = FUNCTION(arg1)
```

No separators (that is, commas) are required since only the first argument is specified. If a function is called using all default argument values or if there are no arguments, the parentheses are still required. For example:

```
%%answer = FUNCTION()
```

Return values

Functions return their answer to a variable. You must assign the return value to a variable in order to access the return value. In the example below, `REVERSE` is the name of the function (which reverses the content of the argument). The argument is a variable called *%%Barcode*. The answer is stored in the original variable, changing the value of that variable.

```
%%Barcode = REVERSE(%%Barcode)
```

Or, you could store the answer as the value of a different variable as follows:

```
%%Newbar = REVERSE(%%Barcode)
```

A function returns its result and also sets three system variables:

- return code *(%%RC*)
- return value (*%%RV*)
- return message (*%%RM*)

> **Note:** If you want to use the *%%RC*, *%%RM*, or *%%RV* system variable anywhere other than directly after the function call, you must set their values to another variable. This is because the value of system variables can change frequently. If a value is assigned to *%%RC* and later in the code you are expecting that value to still be in *%%RC*, it may not be because subsequent function calls or section changes could have modified the value.

Return codes

Return codes, which Enrichment stores in the system variable *%%RC*, are integer values that indicate the success or failure of the function call. While each function may set a specific return code based on its result, all such functions follow these conventions:

- A return code of 0 indicates that the function was successful.
- A negative return code (for example, -1) indicates that invalid arguments were used or a severe error occurred.
- A positive return code (for example, 1) indicates that the function was not successful or was partially successful.

Note: To handle errors that may result from calls to Enrichment functions, you should always include logic to check the return code value. Enrichment does not provide any error messages if a function call fails.

The *%%RC* contains the return code from the most recent function call. As shown in the figure below, you can use *%%RC* to specify alternate processing.

```
%%AMOUNT = FINDNUM(%%LINE1)     /*Find amt in line 1 or 2 */
IF %%RC <> 0 THEN               /* If amount is not found */
  %%AMOUNT = FINDNUM(%%LINE2)
  IF %%RC <> 0 THEN              /*Set to 12 if still not found */
     %%AMOUNT = 12
  ENDIF
ENDIF
```

Return values

Return values, which are integer values that Enrichment stores in the system variable *%%RV*, provide data that supplements the answer returned from the function call. The data stored in the return value (if any) depends on the function. For example, the return value from the FINDZIP function identifies what type of ZIP Code™ was found (ZIP, ZIP + 4®, or ZIP+4+2). Enrichment stores the actual ZIP Code™ in the value returned by the function. As the example below shows, you can use the *%%RV* system variable in the rule file in the same way you use the *%%RC* system variable.

```
%%ZIP = FINDZIP(%%ADDRESS)    /*FIND ZIP CODE, */
IF %%RV = 5 THEN              /*OUTSORT 5 Digit  */
  <OUTPUT> 5DIGIT
ELSE
 <OUTPUT> ZIP4
ENDIF
```

Some functions do not produce return values because they do not produce supplemental results. For example, the `TIME` function does not produce a *%%RV* value.

Return messages

Return messages, which Enrichment stores in the system variable *%%RM*, are C messages that indicate why certain functions failed. For example, if you use the `EXISTS` function on a file that does not exist, Enrichment sets the *%%RM* system variable to the C message "`An I/O abend has been trapped.`"

The figure below shows an example of the *%%RM* system variable in the rule file.

```
IF NOT EXISTS('/usr/lib/myfonttable') THEN
   %%Msg = %%RM
   MESSAGE(1, W, "Could not open font table - " | %%Msg)
ENDIF
```

Operators

The table below lists every Enrichment operator along with its type, a brief description, proper syntax, and processing precedence. Enrichment evaluates mathematical expressions from left to right according to the indicated operator precedence. If a mathematical expression contains parentheses, Enrichment evaluates expressions within the parentheses first.

Note: Enrichment does not support decimal math. You can only perform mathematical operations on whole numbers.

Table 4: Operators

Operator	Types	Description	Syntax	Precedence
()	Any	Open and close parentheses	`(...)`	1
-	Arithmetic	Unary minus	`-%%x`	2
*	Arithmetic	Multiplication	`%%x * %%y`	3
%	Arithmetic	Division (integer)	`%%x % %%y`	3
#	Arithmetic	Remainder	`%%x # %%y`	3
+	Arithmetic	Addition	`%%x + %%y`	4

Operator	Types	Description	Syntax	Precedence
-	Arithmetic	Subtraction	`%%x - %%y`	4
|	String	Concatenate	`%%x | %%y` **Note:** On mainframe systems you can use either a solid vertical bar (`|, EBCDIC X'4F'`) or a broken vertical bar (`¦, EBCDIC X'6A'`) as a concatenation character. On UNIX and Linux systems, the vertical bar is sometimes displayed solid, sometimes broken, but is always ASCII X'7C'.	5
=	Comparison	Equality	`IF %%x = %%y THEN`	6
!= \\= <>	Comparison	Inequality	`IF %%x != %%y THEN`	6
<	Comparison	Less than	`IF %%x < %%y THEN`	6
<=	Comparison	Less than or equal to	`IF %%x <= %%y THEN`	6
>	Comparison	Greater than	`IF %%x > %%y THEN`	6
>=	Comparison	Greater than or equal to	`IF %%x >= %%y THEN`	6

@	Comparison	Checks if the first string is contained in the second string.	`%%x @ "KY TN IN"`
			For example, the statement
			`%%state @ KY CA NC CT`
			is TRUE if *%%state* is any of the following:
			• K
			• KY C
			• KY CA
			• CA
			It is FALSE if `%%state` is:
			• CA KY
			• KY CT
				6

Operator	Types	Description	Syntax	Precedence

@=	Comparison	Use wildcards with this operator to match a pattern. Case sensitive.	`%%x @= "K* T*"`
			For example, the statement
			`%%state @= "K* C*"`
			is TRUE if `%%state` is `"KY CA"`.
			It is FALSE if `%%state` is `"KY NY"`.
			It is also FALSE if `%%state` is `"ky ca"`
				6

*=	Comparison	Use wildcards with this operator to match a pattern. Not case sensitive.	`%%x @= "K* T*"`
			For example, the statement
			`%%state @= "K* C* n* c*"`
			is TRUE if `%%state` is `"KY CA NC CT:.`
			It is FALSE if `%%state` is `"KY NC"`.
				6

NOT	Boolean	Not	IF NOT expression1 THEN	7
AND	Boolean	And	IF expression1 AND expression2 THEN	8
OR	Boolean	Or	IF expression1 OR exression2 then	8

Instructions

Instructions let you control the processing flow of the rule file. There are five types of instructions: IF THEN ELSE, QUIT, FOR…NEXT, DO…LOOP, and SELECT CASE.

IF…THEN…ELSE

When Enrichment encounters an `IF THEN ELSE` instruction, it evaluates the logical expression in the `IF` clause. If it is *TRUE*, then the instructions that follow are executed. If it is *FALSE*, then the first `ELSEIF` logical expression is similarly evaluated. This continues for all `ELSEIF` clauses. If no logical expression is *TRUE*, then the `ELSE` clause, if any, is executed.

A simple IF/THEN/ELSE instruction is shown below.

```
IF logical-expr-1 THEN
   statements-1
ELSEIF logical-expr-2 THEN
   statements-2
ELSE
   statements-3
```

Note: The ELSEIF clause can be omitted or can be repeated as many times as desired. The ELSE clause can also be omitted.

QUIT

The QUIT instruction causes Enrichment to immediately halt execution of the current rule file section. It can be useful for exiting complicated logic when an error occurs. For example:

```
IF %%x = %%y THEN
DO
%%String = Read(DD:FILE)
IF %%String = 'Last line' THEN
  MESSAGE(1, W, “Unexpected last line in :FILE”)
  QUIT
ENDIF
LOOP WHILE %%String = 'Car'
ENDIF
```

SELECT CASE

The SELECT CASE instruction executes one of several groups of statements, depending on the value of an expression. The syntax for SELECT CASE is:

```
SELECT CASE test-expr
CASE expr
statements-1
CASE value TO value
statements-2
CASE [IS] compare-operator expr
statements-2
CASE expr-list
statements-2
CASE ELSE
statements-3
END SELECT
```

where:

- `test-expr` is the expression to compare with each `CASE` expression. This can be a numeric or string expression.
- `if expr` equals the `test-expr`, the case is performed.
- `value TO value` is a range of values to compare with the `test-expr`. This range can be a numeric or string expression. If a string expression is used, message PDR0812W will appear. It may be ignored.
- `compare-operator expr` is a range of values, such as `IS > 10`. The IS keyword is optional.
- `expr-list` is one or more valid case expressions separated by commas. The correct form is:

  `case-expr [, case-expr [, case-expr]...]`

  An example of `expr-list` is:

```
SELECT CASE %%Value
CASE 1
%%Sale = 'single'
CASE 2
%%Sale = 'double'
CASE 3, 5 TO 7, IS > 20
%%Sale = 'special'
CASE ELSE
%%Sale = 'other'
END SELECT
```

  In this example:

  - if the variable *%%Value* is 1, the variable *%%Sale* is set to *single*
  - if *%%Value* is 2, *%%Sale* is set to double
  - if *%%Value* is 3, 5, 6, 7, or greater than 20, *%%Sale* is set to *special*
  - if *%%Value* is anything else, *%%Sale* is set to *other*.

From this example, you can see that the `SELECT CASE` instruction can be used in place of an `IF THEN ELSE` instruction with multiple `ELSEIF` statements.

FOR...NEXT

The `FOR...NEXT` instruction repeats a group of statements a specified number of times.

> **Note:** Infinite loops are possible with `FOR` loops. Enrichment does not check for infinite loops, so you should set time conditions appropriately.

The syntax for `FOR...NEXT` is:

```
FOR %%counter = start-expr TO end-expr [STEP step-expr]
Statements
[EXIT]
[EXIT FOR]
[ITERATE]
[ITERATE FOR]
statements
NEXT [%%counter]
```

where

- `%%counter` is a variable name that will contain the counter value during each loop execution. The `%%counter` variable is set to the `start-expr` before the first loop iteration and is incremented by `step-expr` until it equals or exceeds the value of `end-expr`.
- `start-expr` is the initial numeric value to set the `%%counter` variable prior to the first loop iteration. It may be a simple number or a complex expression.
- `end-expr` is the numeric value that `%%counter` must equal or exceed to terminate the loop execution. It may be a simple number or a complex expression.
- `step-expr` is the numeric value to increment `%%counter` at the end of each loop iteration. It may be a simple number or a complex expression. In addition, it may be either positive or negative.

Hints

- The `STEP` keyword and `step-expr` are optional. The default `STEP` is 1.
- If `start-expr` exceeds `end-expr` at the start of the first loop iteration, the loop will not be executed.
- `EXIT` and `EXIT FOR` both immediately leave a given loop. `EXIT` by itself leaves the innermost FOR or DO loop. Since `EXIT` can also leave a DO loop, `EXIT FOR` is more explicit. For example:

```
FOR %%Count = 1 to 10
DO WHILE %%String != 'Car'
%%String = READ(DD:FILE)
IF %%String = 'Last line' THEN
 EXIT FOR
ENDIF
LOOP
NEXT %%Count
```

  In this example, the `EXIT FOR` statement will leave both the `DO` loop and the `FOR` loop. An `EXIT` alone would have only left the inner DO loop.

  - `ITERATE` or `ITERATE FOR` will cause the next iteration of the loop. The *%%counter* variable is incremented, and the next loop iteration begins at the top. Any statements below `ITERATE` are not executed until the iterations are complete.

- The use of the *%%counter* variable in the `NEXT` statement is optional. Using a variable can help you reference which loop is ending, especially when you have nested FOR loops.
- If you omit the `FOR` on the `EXIT FOR` statement, the `EXIT` means to exit the innermost loop. If you nest a `DO` inside a `FOR`, you can have an `EXIT FOR` statement that will exit the outer FOR loop.
- You can use an `ITERATE FOR` substatement to cause the next iteration of the FOR loop to begin executing immediately. If you specify `ITERATE` by itself, it goes to the top of the innermost loop.

DO...LOOP

The `DO...LOOP` instruction repeats a block of statements while a condition is *TRUE* or until a condition becomes *TRUE*.

> **Note:** Infinite loops are possible with DO loops. Enrichment does not check for infinite loops, so you should set time conditions appropriately.

The syntax for `DO...LOOP` to test the logical expression at the top of the loop is:

```
DO WHILE logical-expr
   statements
LOOP
```

Or

```
DO UNTIL logical-expr
   statements
LOOP
```

The syntax for `DO...LOOP` to test the logical expression at the bottom of the loop is:

```
DO
   statements
LOOP WHILE logical-expr
```

Or

```
DO
   statements
LOOP UNTIL logical-expr
```

where `logical-expr` is the logical expression that will be tested to determine when to leave the loop.

To leave or iterate a DO loop:

```
DO…
statements
[EXIT]
[EXIT DO]
[ITERATE]
[ITERATE DO]
statements
LOOP…
```

Hints

- `WHILE` continues the loop as long as the `logical-expr` is *TRUE*. If the loop is tested at the top and the `WHILE` expression if *FALSE* on the first iteration, the loop will not be executed at all.
- `UNTIL` continues the loop until the `logical-expr` becomes *TRUE*. If the loop is tested at the top and the `UNTIL` expression is *TRUE* on the first iteration, the loop will not be executed at all.
- The loop will always be executed at least once if the loop is tested at the bottom.
- `EXIT` and `EXIT DO` both immediately leave a given loop. `EXIT` by itself leaves the innermost FOR or DO loop. Since `EXIT` can also leave a FOR loop, `EXIT DO` is more explicit. For example:

```
DO WHILE %%String != 'Car'
FOR %%Count = 1 to 10
%%String = READ(DD:FILE)
IF %%String = 'Last line' THEN
EXIT DO
ENDIF
NEXT %%Count
LOOP
```

In this example, the `EXIT DO` statement will leave both the `DO` loop and the `FOR` loop. An `EXIT` by itself would have only left the inner `FOR` loop.

- `ITERATE` or `ITERATE DO` will cause the next iteration of the loop. The `logical-expr` is evaluated and the next loop iteration begins at the top of the loop. Any statements below the `ITERATE` are not executed until the iterations are complete.
- If you omit the `DO` on the `EXIT DO` statement, the `EXIT` means to exit the innermost loop. If you nest a `DO` inside a FOR, you can have an `EXIT FOR` statement that will exit the outer FOR loop.
- You can use an `ITERATE DO` substatement to cause the next iteration of the DO loop to begin executing immediately. If you specify `ITERATE` by itself, it goes to the top of the innermost loop.

Example of Instructions

The following example shows a rule file that contains these instruction groups.

```
<rule>
<content>
DOCUMENT:
%%i = 1
do while %%i < 3
message(0, I, "DO WHILE, i=" | %%i)
%%Mess1 = "DO WHILE, i=" | %%i
WRITE(DDOUTPUT2,%%Mess1)
%%i = %%i + 1
loop
do
message(1, I, "DO UNTIL after, i=" | %%i)
%%Mess2 = "DO UNTIL after, i=" | %%i
WRITE(DDOUTPUT2,%%Mess2)
%%i = %%i + 1
loop until %%i > 3
%%a = substr(%%Account_Number, 1, 4)
%%j = %%a # 20
for %%i = %%j to %%j + 25 step 2
if %%i = 26 then
message(7, I, "Iterating for, i=" | %%i)
%%Mess3 = "Iterating for, i=" | %%i
WRITE(DDOUTPUT2,%%Mess3)
iterate for
endif
select case %%i
case 14
  message(2, I, "Case 1, i=" | %%i)
  %%Mess4 = "Case 1, i=" | %%i
  WRITE(DDOUTPUT2,%%Mess4)
case 16 to 20, 24, is > 33
  message(3, I, "Case 2, i=" | %%i)
  %%Mess5 = "Case 2, i=" | %%i
  WRITE(DDOUTPUT2,%%Mess5)
case is >= 30
  message(4, I, "Case 3, i=" | %%i)
  %%Mess6 = "Case 3, i=" | %%i
  WRITE(DDOUTPUT2,%%Mess6)
case else
  message(5, I, "Case else, i=" | %%i)
  %%Mess7 = "Case else, i=" | %%i
  WRITE(DDOUTPUT2,%%Mess7)
end select
if %%i > 35 then
message(6, I, "Exiting for, i=" | %%i)
%%Mess8 = "Exiting for, i=" | %%i
```

```
WRITE(DDOUTPUT2,%%Mess8)
exit for
endif
next
</content>
</rule>
```

Logical Expressions

Logical expressions result in a value of *TRUE* or *FALSE*. These are represented as integers where 0 is *FALSE* and anything else (usually 1) is *TRUE*. Logical expressions are normally used within IF THEN ELSE instructions. They include:

- Expressions combined with the comparison operators. For a complete list of operators, see **Operators** on page 25. For example:

```
IF %%x = %%y THEN
IF %%x = %%y + %%z THEN
```

- Expressions combined with the Boolean operators. For a complete list of operators, see **Operators** on page 25. The AND and OR connectors combine two expressions, while the NOT connector negates the single expression following it. For example:

```
IF %%w = %%x AND %%y > %%z THEN
```

  (The condition is *TRUE* if both `%%w = %%x` and `%%y > %%z`.)

- A logical function. For example:

```
IF CHANGED(%%y) THEN
```

IF THEN ELSE instructions can include other IF THEN ELSE instructions. These instructions can be nested as deeply as is necessary.

Comparing Numbers

Numbers are always compared as numbers. Field variables are considered numeric if they only contain blanks or numbers. Leading zeros are ignored. Therefore, the string `0005` equals 5.

Comparing Strings

If you do not specify `<CHARACTERS>` in the control file's Environment tag group, Enrichment uses the binary values of characters to compare strings.

- On mainframe systems, characters are represented by the EBCDIC character set. In EBCDIC, the lowercase characters have lower binary values than the uppercase characters.
- On UNIX, Linux, and Windows, characters are represented by the ASCII character set. In ASCII, the lowercase characters have greater values than the uppercase characters. For example, the word cat would compare as shown below.

  Note: Trailing blanks are ignored when comparing strings. Therefore, `"ABC  "` equals `ABC`.

As the following example shows, the EBCDIC hexadecimal value of the word `"cat"` may be `X'8381A3'`, `X'C381A3'`, or `X'C3C1E3'` while the ASCII hexadecimal value of the word may be `X'636174'`, `X'436174'`, or `X'434154'`, each depending on how the word is capitalized.

```
cat < Cat < CAT cat < Cat < CAT
88A C8A CCE 667 467 445
313 313 313 314 314 314

EBCDIC Hexadecimal ASCII Hexadecimal
TranslationTranslation
```

You can use the `<CHARACTERS>` tag to avoid the hexadecimal translation differences and to enable proper comparison of international characters.

Print Stream Commands

Print stream commands let you control document processing from within the rule file. The commands are:

`<APPEND>` Add an insert or input to the current document

`<BANNER>` Add a banner to the current document

`<OUTPUT>` Send the current document to a specified output

For information about these commands, refer to *Enrichment Language Reference Guide* and *Print Stream Commands*.

Comments

Comments are non-process information contained in the control and rule files. Enrichment ignores any text or symbols between the comment begin and end characters. You can nest comments and place them anywhere in the control or rule file to provide an explanation of the code, make notes to yourself or other developers, or to record the date, time, and author of the code.

Comments can be included in any of the following programming styles:

- `/* comment text */`

  Comments can span multiple lines up to the closing */. This is helpful if you need to temporarily comment out a block of lines for testing.

- `<! comment text >`

  Comments can span multiple lines up to the closing >.

- `// comment text`

  This comments the rest of the current line after the two slashes.

All three comment styles will work in both the rule file and the control file.

To comment out blocks of code, use the /**/ style, because the tag syntax <tag> may lead to an incorrect closing of the <!> block of comments.

Note: To comment out a `<GETFILE>` statement you must use a double slash (//). You cannot comment out a `<GETFILE>` statement using other comment markers (such as /*…*/).

Specifying Character Strings

Some tags and syntaxes include parameters with character string values. In most cases, you can specify the string in one or more of the formats listed below. The table lists the valid specification syntax for each format.

Note: If the string contains quotes, you must double the internal quotes or Enrichment assumes they terminate the string.

Table 5: Specifying Strings

Format	Valid Syntax	Notes
Literal String	text or 'text text' or "text text"	If the literal string contains spaces or punctuation other than _ and :, you must enclose it in single or double quotes. If the literal string contains extended ASCII characters 0x80 - 0xFF, you must enclose it in single or double quotes. For mainframe, literal strings are stored in EBCDIC format. For UNIX, Linux, and Windows, literal strings are stored in ASCII format.
ASCII	A'string' or a'string'	
Binary	B'string' or b'string'	Binary strings can contain only the characters 0 and 1.
EBCDIC	E'string' or e'string'	
Hexadecimal	X'string' or x'string'	Hexadecimal strings can contain only the numbers 0 through 9 and the letters A through F.
Symbolic	s'string' or s'x=# y=#'	A symbolic string can be any mnemonic AFP structured field ID. Space can be a delimiter in 'x=# y=#'. You can insert 'x=# y=#' or just one of the values.

Specifying Files

To specify a file in Enrichment you use the conventions of the operating system as described in the following sections.

Specifying Files on Mainframe Systems

Mainframe file names can be up to 55 characters in any of the following formats:

- *<tag>* DD:*ddname*
- *<tag>* DD:*ddname(member)*
- *<tag>* '*qualifier.qualifier.qualifier*'

- *<tag>* '*qualifier.qualifier.qualifier(member)*'

Where:

- *ddname* is the data definition of the file in your JCL
- *member* is the member name of a PDS
- *qualifier* is a node of a data set

Alternatively, mainframe file names can be specified in this format:

```
<tag> JES:DD:ddname
```

Where *ddname* is the data definition of the file in your JCL. To access the JES2 spool use `<FILE> JES:DD:ddname`. The *ddname* contains one record to identify the spool file in the following comma-delimited format:

JobNumber,JobName,StepName,ProcName,DDName

Where:

- *JobNumber* is the job that created the JES file
- *JobName* is the name of the job that created the JES file
- *StepName* is the name of the job step that created the JES file
- *ProcName* is the name of the procedure used, if applicable
- *DDName* is the name of the DD that specifies the JES file

You can omit *ProcName* by placing two commas after *StepName*.

> **Note:** In rule files, quotation marks indicate a word or phrase that contains blanks or special symbols. Therefore, when using mainframe file specifications in rule files, you must surround fully-qualified file names with double and single quotation marks so that the single quotation marks will be kept with the file specification.

Specifying Files On UNIX and Linux

On UNIX and Linux, make directory and file specifications in Enrichment control files as follows.

```
<tag>/dir1/K/dirn/filename
```

Where <*tag*> is the Enrichment control file tag or rule file language component that precedes the file specification.

All directory specifications (dir1, dir2, and so on) and file names can be up to 255 characters in length.

> **Note:** UNIX and Linux Enrichment users can utilize pipes to point input data to different outputs without changing the control file.

If you do not begin the file specification with a slash (/), Enrichment assumes the file specification is relative to your home directory. If you begin the file specification with a slash (as shown in the example), Enrichment assumes the file specification is relative to the root directory. UNIX and Linux file and directory names are case-sensitive. Therefore, the directory or file "Accounting" is different from the directory or file "accounting".

If you use quotes to specify a file name, Enrichment will take whatever is between the quotes as the filename.

Specifying Files On Windows

On Windows, make directory and file specifications in Enrichment control files as follows:

```
<tag> d:\dir1\K\dirn\filename
```

Where *<tag>* is the Enrichment control file tag or rule file language component that precedes the file specification:

- To specify directories and files that include spaces in their names, enclose the entire directory and file name in quotes. You can use either single quotes (') or double quotes ("). Any tag that takes a file name as a parameter can have the name enclosed in quotation marks, even if there are no spaces in the directory or file name.
- The drive specification (d: in the example above) is optional. If you omit the drive specification, your current drive is assumed.
- The total size of all directory names and the file name can be up to 255 characters.

> **Note:** On Windows, you can use pipes to point input data to different outputs without changing the control file.

If you do not begin the file specification with a backslash (\), Enrichment assumes the file specification is relative to your current directory. If you do begin the file specification with a backslash, Enrichment assumes the file specification is relative to the root directory.

Pipes for UNIX, Linux, and Windows Users

UNIX, Linux, and Windows users can utilize pipes to redirect input to different outputs (such as to the screen, to an output file, or to another program) without changing the control file.

> **Note:** | is the pipe symbol.

- If the `<FILE>` tag used in the `<INPUT>` group is `<FILE>|` the data will be read from standard input.

- If the `<FILE>` tag used in the `<OUTPUT>` group is `<FILE> |` the data will be written to standard output.

An example of a control file that uses pipes is shown below.

```
<!   Rule File: pipe.ctl                                    >
<!   Author:    Precisely                                >
<!   Purpose:   How to use pipes for input and output.      >
<INPUT>
   <NAME> INPUT_FILE
   <FILE> |
   <TYPE> P
   <DOCUMENT> 1
   <FIELD> %%NAME R
      <LOC> 2 1 3
      <REPLACE>* YES
   </FIELD>
</INPUT>
<RULE>
   <CONTENT>
      IF %%NAME='JIM' THEN
      %%NAME='JAMES'
      ENDIF
   </CONTENT>
</RULE>
<OUTPUT>
   <NAME> OUTPUT_FILE
   <FILE> |
</OUTPUT>
```

Specifying Measurements

Some tags include the units parameter. Unless otherwise indicated, units is an optional parameter that defaults to IN (inches). Enrichment recognizes the following units values:

- IN (Inches)
- MM (Millimeters)
- CM (Centimeters)
- POINTS (Points)
- PELS (Pixels)

Measurements for which you set units to IN, MM, and CM can contain up to three decimal places. You must specify measurements for POINTS or PELS as integers, except for PostScript where the measurement may contain a decimal value.

If you specify the measurement in anything other than pels, Enrichment converts the measurement to pels based on the value in the `<DENSITY>` tag.

> **Note:** If you do not specify a units value on a tag for which you specified a measurement, Enrichment assumes the measurement is in inches.

Regardless of the units value, Enrichment processing messages indicate most measurements in pels.

- In AFP environments, 240 pels typically equals one inch
- In DJDE environments, 300 pels typically equals one inch
- In PCL environments, 300 pels typically equals one inch
- In PDF environments, 72 pels typically equals one inch
- In PostScript environments, 300 units typically equals one inch

> **Note:** In all environments, the `<DENSITY>` tag can affect or set how many pels equal one inch.

3 - Developing an Application

This section describes the high-level process of developing an Enrichment application.

In this section

Application Development Process

This section describes the high-level process of developing an Enrichment application.

Necessary Skills

Typically, one person can fully implement an Enrichment application. Factors that can affect the success of the development effort include the complexity of the application, the required and available resources, and the expertise of the application developer.

In addition to Enrichment proficiency and depending upon the application itself, the developer may require expertise or assistance in the use of inserter machines, postal regulations, address cleansing and postal presort software, print resources, document design, and supplemental data file creation.

Identifying User Requirements

To develop an application, you need to know the user requirements. Depending upon the complexity of the application, you may want to write a formal functional specification. The key tasks in defining a functional specification for an Enrichment application are:

Identify Print Stream Types

Enrichment fully supports impact, AFPDS, AFP line-data, AFP mixed data, Metacode, DJDE, PCL PDF, and PostScript print streams. Limited Enrichment processing may be possible with other print stream types. The procedures for extracting data from and adding data to documents vary depending on the type of print stream.

Identify Printer Types

Capabilities for using fonts, drawing barcodes, and doing overlays vary among printers. Before starting an application, you must understand the printers used and their limitations. If you combine print streams from separate applications, the printer must be compatible with each of the print stream formats used. For example, you cannot combine AFP print streams with DJDE print streams without first performing a process (external to Enrichment) that converts the AFP print streams to DJDE or vice versa.

Identify Input Processes

Enrichment is capable of performing a host of functions. Generally, these fall into the categories below. You should identify all functions your Enrichment application is to perform before development begins.

- Add, move or delete text
- Add, move or delete barcodes
- Standardize addresses according to CASS™ regulations
- Add ZIP + 4® and POSTNET™ barcodes
- Consolidate multiple documents into a single document (for example, consolidate all documents going to the same address)
- Merge multiple print streams into a single print stream (commingling)
- Add overlays to a document
- Access a relational database for queries or updates
- Add electronic inserts and banner pages to a document or build custom documents
- Personalize documents with variable values
- Use postal Presort or other software to sort documents within a print stream
- Reorder the pages within each document
- Convert to multiple-up or duplex formats
- Divide the documents into multiple outputs at user- or system-defined points
- Route each document to one or more outputs
- Create one or more extract files or report files
- Create a Reprint Index for use with Reprint
- Create side files for use as Mail Run Data Files (MRDFs) for finishing systems
- Add color to the document (AFP, Metacode, PCL, and PostScript print streams only)

Preliminary Design

The next step in developing an Enrichment application is to identify which Enrichment functions are necessary to meet the user requirements. As you define the functions to use, you should familiarize yourself with the input print streams you will process so that you can find the information Enrichment requires.

Key tasks in defining the functional requirements include identifying the following:

New Page Indicators in Each Document

Enrichment can automatically identify page breaks for most print stream types. However, if you work with non-AFPDS inputs that have non-standard carriage controls, contain more than one logical page on each side of a physical page, or use record counting to determine pages, you must identify some criterion by which Enrichment will recognize the top of each page. Typically, you can use a field that contains data unique to the top of each page (for example, a customer number).

New Document Indicators

If your input print streams contain more than one document, you must identify some criterion by which Enrichment will recognize the top or bottom of each document. Typically, you can use a field that contains data unique to the first or last page of each document (for example, a customer number), or—if each document is a specific number of pages in length—you can specify that number.

Note: You can specify multiple top- or bottom-of-document criteria. Refer to the <DOCUMENT> tag discussion in the *Enrichment Language Reference Guide* for more information.

Note: You can specify multiple top- or bottom-of-document criteria. Refer to the <DOCUMENT> tag for more information.

Document Data Fields

You should identify data to extract from the documents. If you process multiple types of documents, you should indicate the data extraction process for each.

Finishing Barcode Contents

If you want to add barcodes to documents, you must identify the data items to use to create the barcodes and the location of the barcodes on the page. If the barcode does not go on every page, specify the pages on which to place it. If you use multiple finishing systems, specify the barcode contents and positions for all possible cases.

Added Text Contents

If you want to add text to documents, identify how Enrichment should select and develop the text. Indicate the font and color to use and the text position. If the text does not go on every page, identify the pages on which to place it.

Address Certification Process

If you want to CASS™-certify (cleanse) addresses, specify the cleansing software to use. Designate the processes for handling corrected, invalid, and uncorrectable addresses.

Consolidation Fields

If you want to consolidate multiple print streams into common envelopes, designate the data items to match (such as customer number, customer name, address, and so on).

Banner Pages, Electronic Inserts, and Overlays

If you want to add banner pages, inserts, or overlays to documents, specify the file names and conditions under which to add them. Note if the insert is to be assembled in a particular position or order within the document. Banners and inserts can contain variable “fill-in-the-blank” items to be completed by Enrichment. Identify any such variables to be processed in the inserts or banners.

Note: You must prepare banner pages, inserts, and overlays outside Enrichment. They should be in a format consistent with the print stream to which they will be added.

Variable Items to Replace

If you want to personalize documents, identify the variables to replace and the source of the variable data.

The Postal Presort Process

If you want to perform postal presort, identify the presort software to use.

Sort Criteria

If you want Enrichment to change the order of the documents, specify the sort criteria.

Page Order

If you want Enrichment to change the order of the pages in a document, specify the reorder sequence.

Extract File Contents

If you want Enrichment to create a side file, specify the data items to extract and their positions in the file.

AFP Index Contents

If you want Enrichment to create an AFP Index, specify the data items to use in it.

Reprint Index Contents

If you want Enrichment to create a Reprint Index, specify the data items to use to select documents for reprinting.

Inserter Control Data File Contents

If you want Enrichment to create inserter control data files, specify the information to record for each document.

The Output Files to Create

Designate each output. If you want Enrichment to create more than one output, specify the criteria for routing documents to each output. If an output has a maximum size threshold (total documents), specify the criteria for breaking the output.

Building the Application

Generally, you will use the following process to develop an Enrichment application.

Note: You can use Enrichment Visual Engineer to interactively develop the application. Visual Engineer is a development tool that enables both IT and document production staff to analyze print streams and build and test Enrichment applications.

Step 1: Create a control file for the application

For instructions on creating a control file, refer to **Creating a Control File** on page 75. Enrichment can perform most functions and combinations of functions in one pass. However, it may be convenient or necessary to run Enrichment in multiple passes. In this case, each Enrichment pass requires a separate control file.

Step 2: Create a rule file

If you want Enrichment to perform conditional processing, you must create a rule file. For more information, refer to **Creating a Rule File** on page 81.

Step 3: Write or identify any external functions

If you want Enrichment to call any external functions, write them or identify which existing functions will be called. For more information, refer to **Using User-Written Functions** on page 89.

Step 4: Create supplemental data files

If the documents being processed do not contain all of the necessary customer data, create a variable data file and specify it as an input. You can also use data files to perform case-driven document assembly if you are not using input print streams. In this case you must create a data file containing all pertinent customer information.

Step 5: Create electronic inserts or banners

If you want Enrichment to add electronic inserts or banners, you must first design and create them. You should store electronic forms and overlays in the mainframe, UNIX, or Windows file system.

Step 6: Create printer resources if necessary

You may need to create and store forms, overlays, and fonts (including POSTNET™ and Code 3of9 fonts).

Note: Enrichment does not normally create or manipulate printer resources associated with print streams. Using, <RESOURCESCAN> and <RESOURCEOUTFILE>, Enrichment can modify inline AFP resources and export them.

- Xerox resources are stored on the printer's fixed disk.
- AFP resources are stored in libraries on the system.

Testing the Application

You should set up the application initially as a stand-alone process. Copy sample input print streams into a test area for application development, then modify your script so that it includes all files and Enrichment modules necessary for the application.

For information about running your application, refer to:

- **Running on Mainframe Systems** on page 201
- **Running on UNIX** on page 205
- **Running on Windows** on page 207

See also: **Troubleshooting an Application** on page 219 and **Testing Performance** on page 221.

Moving the Application to Production

After you successfully test your application as a stand-alone process, you can move it to production. To move the application to production:

1. Modify the script for the existing process to include the appropriate Enrichment steps.
2. Include the stand-alone script for the Enrichment steps in the job schedule.
3. Follow your change control procedures to migrate the updated script into production.

Processing Flow

Enrichment processes your application in a specific manner depending on the types of print stream manipulations that your application performs. Enrichment automatically picks the method based on the functions being performed in the application. There are three types of processing that can occur:

- One-at-a-time processing
- All-at-a-time processing without presort
- All-at-a-time processing with presort

It is critical that Enrichment application developers fully understand these processes to code effectively. The key points for application developers to remember are:

- The order of the steps is important, especially when processing occurs for rule file sections.
- Some steps create data that is saved in system variables (non-user-defined variables set by Enrichment). For example, the CASS step produces variables that contain the cleansed address lines. You should not use system variables in Enrichment steps that occur prior to the step in which the system variable is populated with a value (so, you should not use CASS system variables in the START section of the rule file, for example).
- In both all-at-a-time processing methods, Enrichment reads and stores all documents in memory before processing them. This requires more memory and I/O than one-at-a-time processing.

Processing Steps

The steps in Enrichment processing are listed below. Not all steps are performed for each of the three processing types. See the diagrams in **Process Flow Diagrams** on page 52 to find out which steps are performed with each type of processing.

Validate Control File and Compile Rules

When you run Enrichment, it first conducts a complete scan of the control file and rules to ensure their validity.

Validation for the control file includes checking that all tag groups begin and end properly, are nested properly (when appropriate), and contain properly coded tags. Control file validation also checks for the existence of all named files. Enrichment then adds any messages returned from the validation process to the Enrichment Report file. Any severe error conditions halt processing when they occur.

Rule file compilation includes syntax checking for all rules and functions, as well as confirmation that named files exist on the system. Again, Enrichment records error conditions in the Enrichment Report file and halts processing if a severe error occurs.

Run START: Rules

Enrichment initializes variable values. The `START:` section of the rule file runs once per application.

Read and Analyze One Document

Enrichment performs any print stream analysis tasks coded into the control file. Typical print stream analysis tasks include:

- Identifying individual documents and pages within the print streams
- Identifying data to be read or extracted from the print streams
- Finding address information within the print streams

CASS™ Cleanse Address

If the control file contains a CASS tag group, Enrichment passes addresses to the specified address hygiene software. The cleansing software returns a new (cleansed) address and certain return codes.

Store Document

The contents of each document and its associated variables are temporarily stored for future access.

Sort Documents by ZIP Code™ and Call a CASS™ tool

If the control file contains a CASS tag group and the <DOUBLESORT> tag is set to YES, Enrichment performs <DOUBLESORT> tag functions to sort print data by ZIP Code™, then uses the specified address hygiene software to cleanse addresses.

Sort Documents by <SORTMATCH>

If the control file contains a Sortmatch tag group, Enrichment sorts documents within the print streams into the specified order.

Match Multiple Inputs

If matching is specified in the Sortmatch tag group, Enrichment matches documents appropriately. For additional information, refer to the section on the Sortmatch tag group in *Enrichment Language Reference Guide*.

Retrieve One Document from Storage

The contents of each document are read from temporary storage.

Run DOCUMENT: Rules

Enrichment processes rules, built-in functions, and user-written functions in the `DOCUMENT:` section of the rule file. Enrichment processes `DOCUMENT:` rules once for each document.

Add Electronic Inserts

If the control file contains the Insertrec and/or Insertpage tag groups, Enrichment inserts the appropriate information at the designated locations in the documents.

Write Presort Record to <SORTFILE>

If the control file contains a Presort tag group, Enrichment writes a record composed of Presort sort parts to the Presort Index file.

Presort

As appropriate, Enrichment calls a program to presort documents in the output print streams. Enrichment can call any external program including a sort program.

Run PRESORTED: Rules

Enrichment processes `PRESORTED:` section rules for each document the presorted output.

Prepare Output Document

Enrichment prepares the output print streams according to Output group tagging.

Run PAGE: Rules

Enrichment processes the `PAGE:` section rules once for each page in each output document.

Write One Page to Output

With all other processing complete, Enrichment writes the document page to the output file.

Write Sidefile Record or Index

Enrichment creates any index files (such as the AFP Index or the Reprint Index) or side files defined in the control file.

Run FINISH: Rules

Enrichment processes the `FINISH:` section rules (write reports, banners, and so on). Enrichment processes `FINISH:` rules once per application.

Summarize Processing

Enrichment summarizes the entire processing run in the Enrichment Report file.

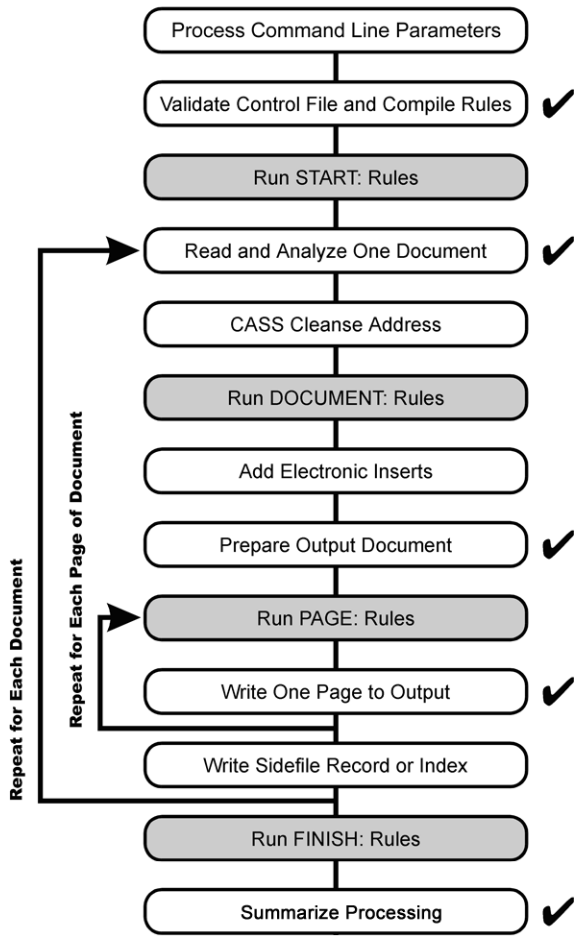
Process Flow Diagrams

The diagrams below illustrate the three types of Enrichment processing. Steps that are required for every Enrichment application are indicated with a check mark. The boxes shaded in gray indicate specific parts of the rule file.

Some steps are not represented in these figures. For example, banner pages can be added in several places with the rule file. Also, not all of the steps represented are required. For example, if you do not have a CASS tag group in your control file, Enrichment does not call CASS™ software to standardize the addresses.

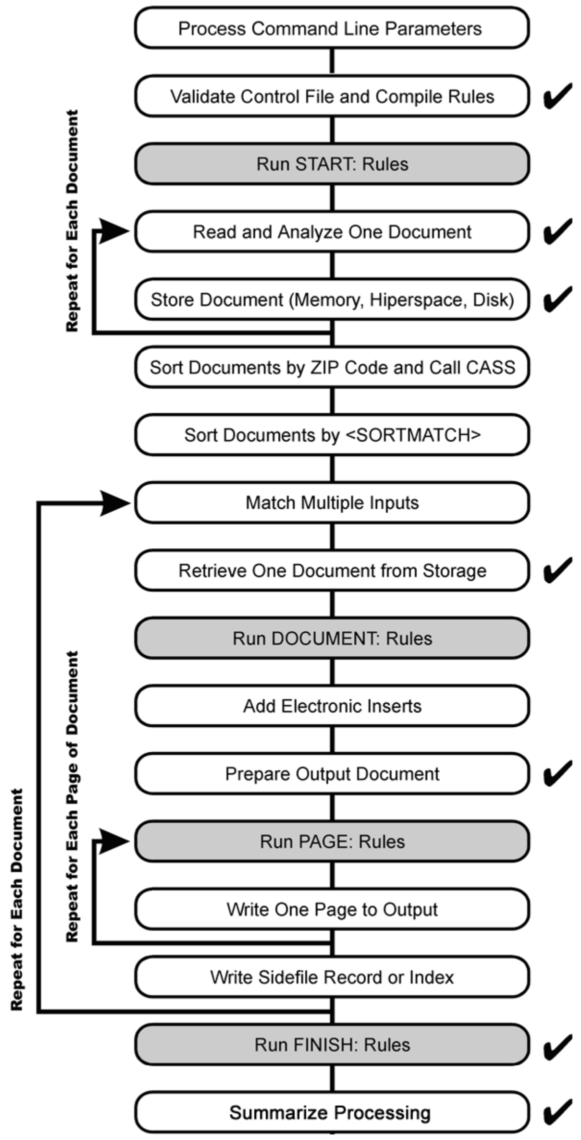
One-at-a-Time Processing

One-at-a-time processing means that Enrichment completely processes one document in the input before moving to the next document. Enrichment uses one-at-a-time processing when document order in the print stream does not need to be altered (that is, when you do not need to sort, match or double sort documents within an input). The diagram below illustrates the one-at-a-time processing flow.


Process Command Line Parameters
Validate Control File and Compile Rules ✔
Run START: Rules
Read and Analyze One Document ✔
CASS Cleanse Address
Run DOCUMENT: Rules
Add Electronic Inserts
Prepare Output Document ✔
Run PAGE: Rules
Write One Page to Output ✔
Write Sidefile Record or Index
Run FINISH: Rules
Summarize Processing ✔
Repeat for Each Document
Repeat for Each Page of Document

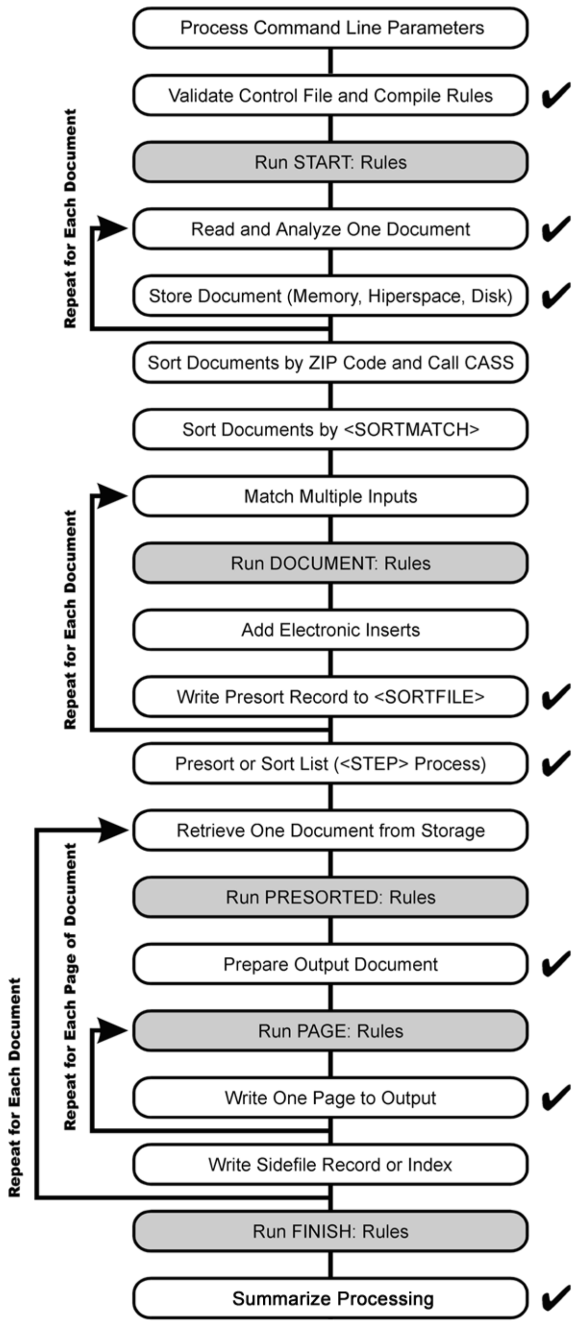
All-at-a-Time Processing without Presort

All-at-a-time processing without presort occurs when you modify the document order (sort, match or double sort) but do not perform presort processes on the output. The diagram below illustrates the processing flow for all-at-a-time processing without presort.


Process Command Line Parameters
Validate Control File and Compile Rules ✓
Run START: Rules
Read and Analyze One Document ✓
Store Document (Memory, Hiperspace, Disk) ✓
Repeat for Each Document
Sort Documents by ZIP Code and Call CASS
Sort Documents by <SORTMATCH>
Match Multiple Inputs
Retrieve One Document from Storage ✓
Run DOCUMENT: Rules
Add Electronic Inserts
Prepare Output Document ✓
Run PAGE: Rules
Write One Page to Output
Repeat for Each Page of Document
Write Sidefile Record or Index
Repeat for Each Document
Run FINISH: Rules ✓
Summarize Processing ✓

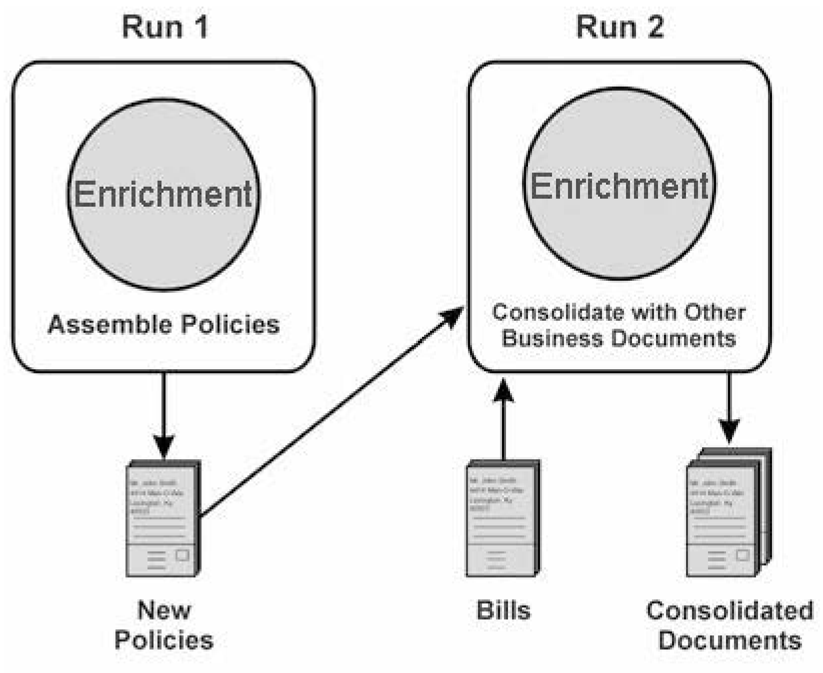
All-at-a-Time Processing with Presort

All-at-a-time processing with presort occurs when you modify the document order (sort, match or double sort) and presort processes are performed on the output. The diagram below illustrates the flow for all-at-a-time processing with presort.


Process Command Line Parameters
Validate Control File and Compile Rules ✔
Run START: Rules
Read and Analyze One Document ✔
Store Document (Memory, Hiperspace, Disk) ✔
Repeat for Each Document
Sort Documents by ZIP Code and Call CASS
Sort Documents by <SORTMATCH>
Match Multiple Inputs
Run DOCUMENT: Rules
Add Electronic Inserts
Write Presort Record to <SORTFILE> ✔
Repeat for Each Document
Presort or Sort List (<STEP> Process) ✔
Retrieve One Document from Storage
Run PRESORTED: Rules
Prepare Output Document ✔
Run PAGE: Rules
Write One Page to Output ✔
Repeat for Each Page of Document
Write Sidefile Record or Index
Repeat for Each Document
Run FINISH: Rules
Summarize Processing ✔

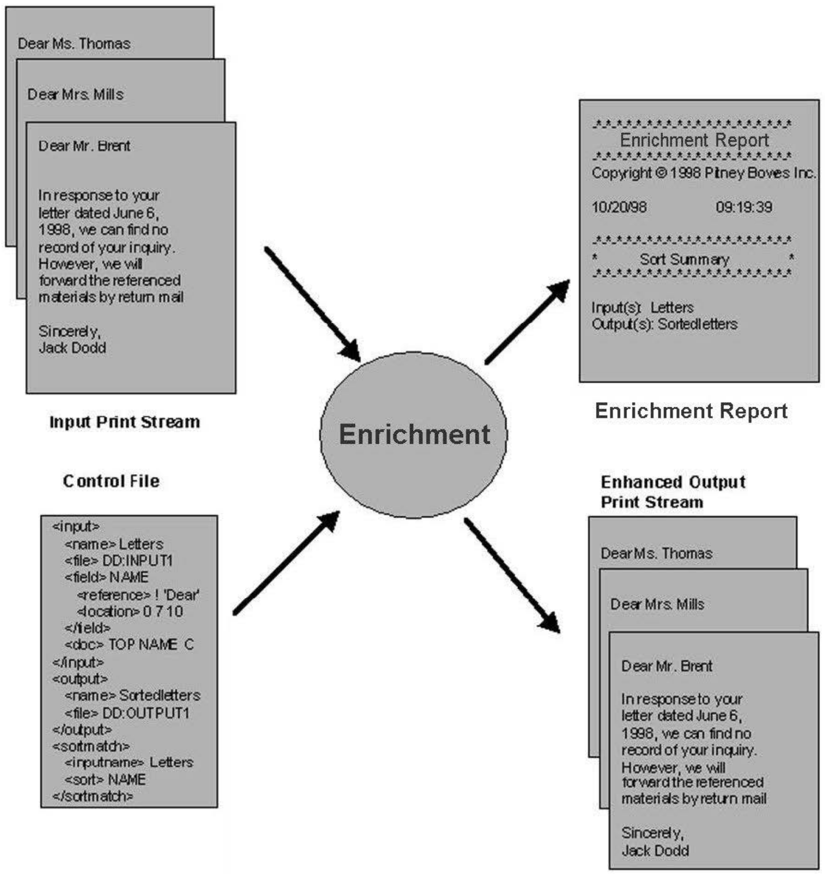
Using Multiple Enrichment Runs

Enrichment can perform many functions in a single pass. However it may be more convenient to process some applications using more than one Enrichment run. In fact, some complex applications may require that you run Enrichment more than once. The following shows a simple example in which two Enrichment runs create customized documents and combine these documents with the output from another business application.



Input and Output

Enrichment requires at least two files for input: a control file (the Enrichment application code) and an existing print stream file. From these, Enrichment creates two or more output files: the Enrichment Report and an enhanced print stream. The diagram below illustrates the minimum input and output for an Enrichment application.



In response to your letter dated June 6, 1998, we can find no record of your inquiry. However, we will forward the referenced materials by return mail

Sincerely,
Jack Dodd

```
********************
  Enrichment Report
********************
Copyright © 1998 Pitney Bowes Inc.
10/20/98          09:19:39

********************
*    Sort Summary   *
********************

Input(s): Letters
Output(s): Sortedletters
```

```
<input>
  <name> Letters
  <file> DD:INPUT1
  <field> NAME
    <reference> ! 'Dear'
    <location> 0 7 10
  </field>
  <doc> TOP NAME C
</input>
<output>
  <name> Sortedletters
  <file> DD:OUTPUT1
</output>
<sortmatch>
  <inputname> Letters
  <sort> NAME
</sortmatch>
```

In response to your letter dated June 6, 1998, we can find no record of your inquiry. However, we will forward the referenced materials by return mail

Sincerely,
Jack Dodd

Optional inputs to Enrichment include:

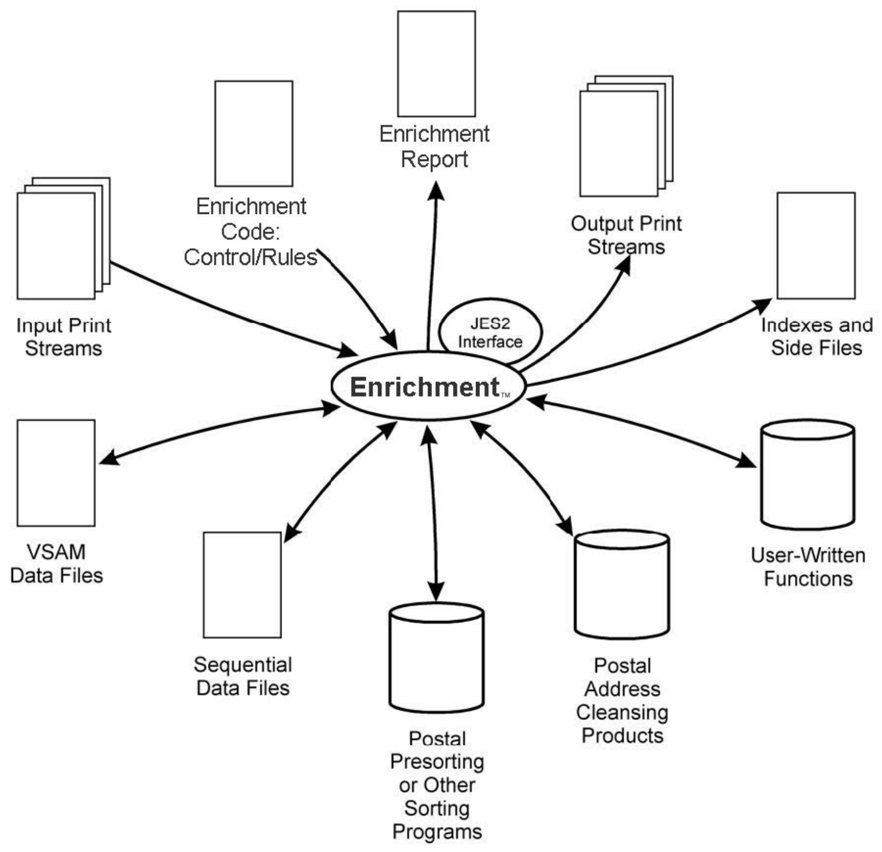
- Multiple input print streams to combine
- Supplementary information, including inserts and banners, to use as added pages or added records
- A rule file for conditional processing
- Postal presort parameter files
- An address database
- (Mainframe only) Temporary Virtual Storage Access Method (VSAM) file for Enrichment I/O processing

Optional outputs from Enrichment include:

- Multiple enhanced output print streams, output page ranges, or convenience breaks
- Multiple side files per output, including Reprint Indexes and AFP Indexes, that contain user-specified information about each document in the output
- Postal address cleansing and presort reports can be included in Enrichment output when an address cleansing or postal presort product (such as Finalist or Mailstream Plus) is called from Enrichment

Enrichment can also read and write external sequential or VSAM files and can access user-written functions.

The following diagram fully illustrates the input and output types.



Specifying an Input Print Stream

An Input tag group defines one print stream to process in your application. You must place one Input tag group in the control file to identify each print stream. The Input tag group begins with the <INPUT> tag and ends with the </INPUT> tag. Neither tag has parameters.

The following tags are used to define the basic attributes of a print stream. Refer to the *Enrichment Language Reference Guide* for more information on these tags and for a complete listing of all Input tag group tags.

- `<FILE>` The path and file name of the input print stream.
- `<NAME>` is a unique reference name for the input print stream. You use this reference name to identify the input in other Enrichment tags. The `<NAME>` tag value cannot contain spaces. For example, if your input print stream contains first quarter dividend information, you might code the `<NAME>` tag as follows:

  ```
  <NAME> DIVIDENDS_1Q2006
  ```

- `<TYPE>` identifies the format of the input print stream. For example, AFP line data, DJDE, Metacode, etc.

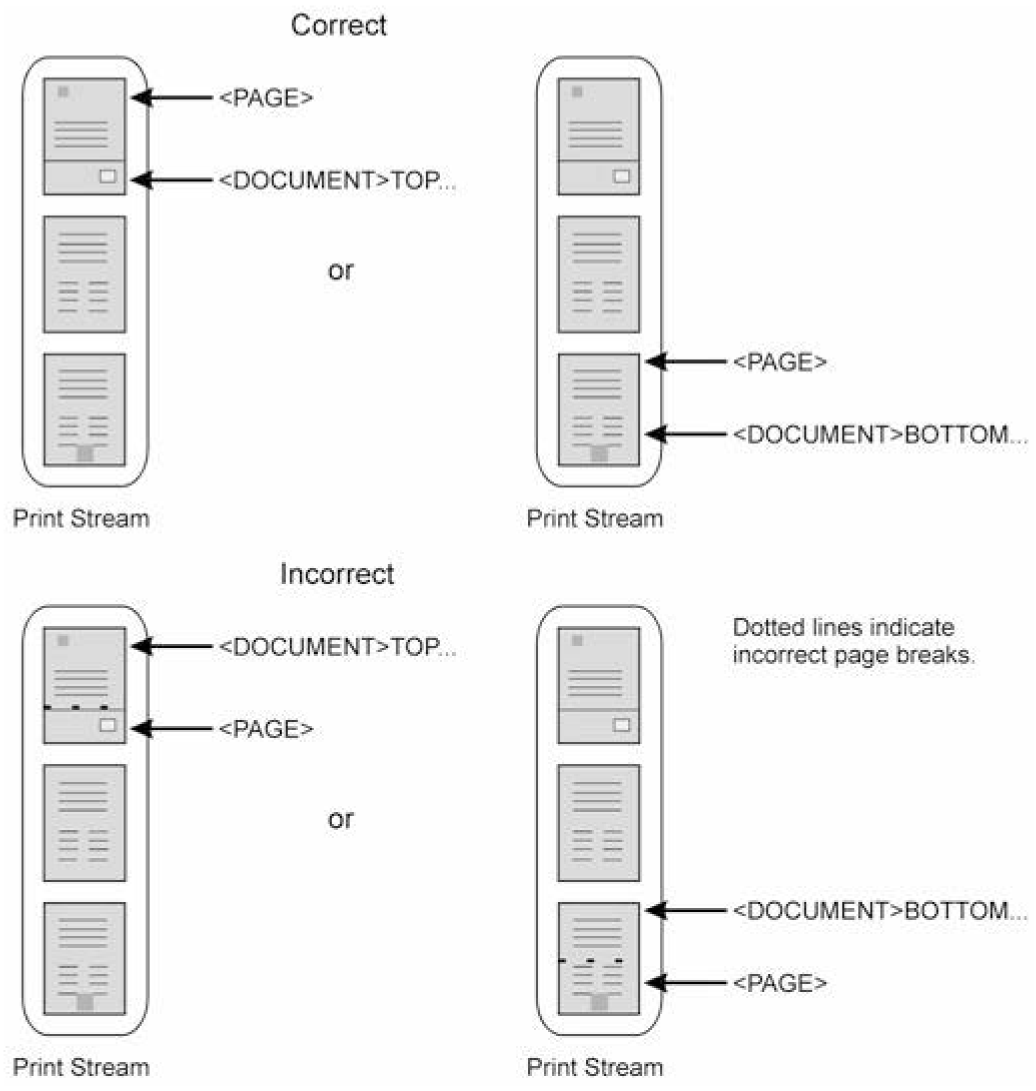
- `<RECORD>` If you are developing applications to run on UNIX, you must use the `<RECORD>` tag in your Input group to identify how to read the input print stream. The `<RECORD>` tag defines the characteristics of record length indicators in an input or insert so that UNIX can properly identify individual records.
- `<HEADER>` The Input group `<HEADER>` tag identifies a number of header lines to skip in the input print stream before the first document and whether to keep the header information when the output stream is written (that is, whether to add the header information to the top of the output print stream). Within Metacode print streams, the `<HEADER>` tag is often used to capture the DJDE printer and job setup instructions that occur before the first document. Refer to the *Enrichment Language Reference Guide* for a complete discussion of the `<HEADER>` tag.
- `<IDEN>` specifies the criteria by which Enrichment recognizes DJDE records in a print stream.

Defining Page Breaks and Document Breaks

To properly process your print stream you need to indicate where one document stops and another starts. You may also define where the page breaks are within each document, or you can let Enrichment identify page breaks automatically.

A common mistake in print stream analysis is defining the top or bottom of a document somewhere other than the first or last page of the document. This mistake is especially common when the user specifies a `<PAGE>` tag and Enrichment does not determine pages automatically. As the following figure illustrates, the control file must meet two conditions to ensure proper document and page definition:

- The first `<PAGE>` tag location (or the first automatic page break) must occur on the first record of the first document in the print stream.
- The `<DOCUMENT>` tag location that identifies top-of-document must occur before the condition that causes page 2 (that is, it must occur on page 1). If you are defining bottom-of-document, the `<DOCUMENT>` tag location must occur on the last page of the document.



Defining Page Breaks

Enrichment can automatically identify pages in most types of print streams (that is, it can identify pages without using tags). It does so by looking for an ANSI carriage control of 1 or a machine carriage control of X8B or X89 in column 1.

> **Note:** You must use a Field group `<REFERENCE>` tag to locate any field used to specify top-of-page since it cannot be located from the top of the page (because the page is not yet defined).

If you do not want Enrichment to automatically determine page breaks in your print stream (for example, if you have multiple logical pages all of which use carriage control 1 in column 1 on one physical page), you must use a `<PAGE>` tag to identify top-of-page. For more information, see the *Enrichment Language Reference*.

Defining Document Breaks

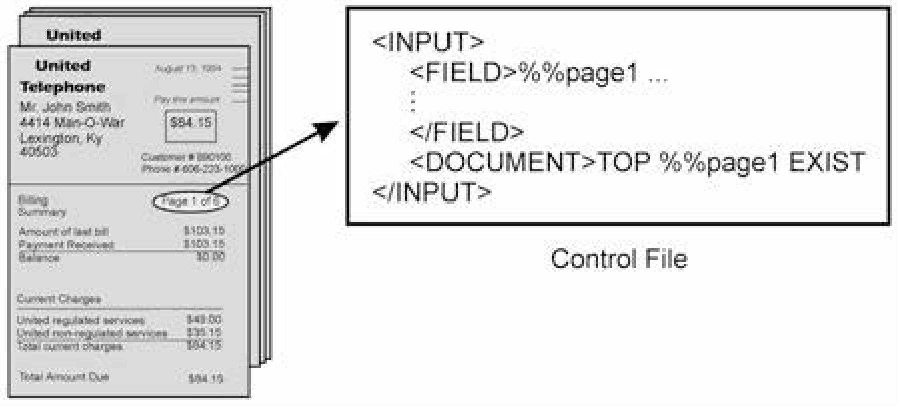
An input can contain one or more documents. A document is a group of pages to be sent to the same customer or destination. Enrichment processes the pages of a document together.

> **Note:** Enrichment allows multiple `<DOCUMENT>` tags in an Input group. If an input contains multiple types of documents, you can specify multiple `<DOCUMENT>` tags to locate each document type.

If your inputs contain more than one document, you must identify some criterion by which Enrichment will recognize the first or last page of each document. If each document contains a specific number of pages, you can specify that number. Or, you can use a field that contains data unique to the first or last page of each document (for example, a customer number).

Use the `<DOCUMENT>` tag to identify documents in an input. For more information, see the *Enrichment Language Reference Guide*.

In the following figure, a `<FIELD>` tag is used to watch for an occurrence of Page 1 in a document. The `<DOCUMENT>` tag tells Enrichment that Page 1 indicates the first page of each document. The specified field may be anywhere on the page.



```
<INPUT>
   <FIELD>%%page1 ...
   :
   </FIELD>
   <DOCUMENT>TOP %%page1 EXIST
</INPUT>
```

Control File

Identifying Fields on a Document

Data items on a document that Enrichment will use or update, such as the customer name, are called "fields". With the exception of rules, fields are the most important and most versatile tool you use to develop Enrichment applications. Fields define information to extract from the document for use in sorting, adding objects, creating side files, finding address information, creating the presort index, defining top or bottom of document and top of page, and in rule files.

Use the Field tag group to specify the instructions for processing a field. Generally, these instructions tell Enrichment to extract a data item into a specified variable and, optionally, to update the print stream if the value of that variable subsequently changes. The Field tag group also identifies how to locate and extract the data.

Since a field is part of a print stream, the Field tag group belongs in an Input tag group. You must define one Field tag group for each data item to select. A number of tags comprise the Field group, but the tags required in every Field group definition are:

- `<FIELD>` specifies a variable name for the field, actions to take on the field, and how to handle leading and trailing spaces.
- `<LOCATION>` defines where the field is located within the document.

Defining a Fixed Position Field

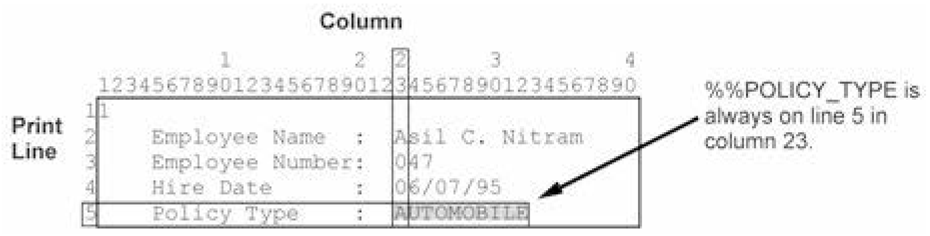
The easiest way to define a field is to specify its name and location using the syntax shown in the following example.

> **Note:** The method shown below is valid only for line data print streams. You cannot specify print positions for AFPDS or Xerox Metacode data.

```
<FIELD> fieldname
    <LOCATION> row column length
</FIELD>
```

You can use this method when the field information always occupies the same position on the printed page.

For example, assume you have an impact print stream in which you must determine what kind of insurance policy each client receives so you can outsort each policy based on type. As shown in the following example, the policy type information is always in column 23 of the fifth line on the page. The longest policy type is 10 characters in length (`AUTOMOBILE`).



```
           1         2 2       3         4
  1234567890123456789012345678901234567890
1 1
2      Employee Name   :  Asil C. Nitram
3      Employee Number:  047
4      Hire Date       :  06/07/95
5      Policy Type     :  AUTOMOBILE
```

In this case, you could identify the Field group as shown in the following example.

```
<FIELD> %%Policy_Type
   <LOCATION> 5 23 10
</FIELD>
```

Defining a Reference Field

When data does not appear in the same position on the printed page but is relative to another piece of unique data on the page, you can use the `<REFERENCE>` tag to define a reference point. The `<REFERENCE>` tag defines the following:

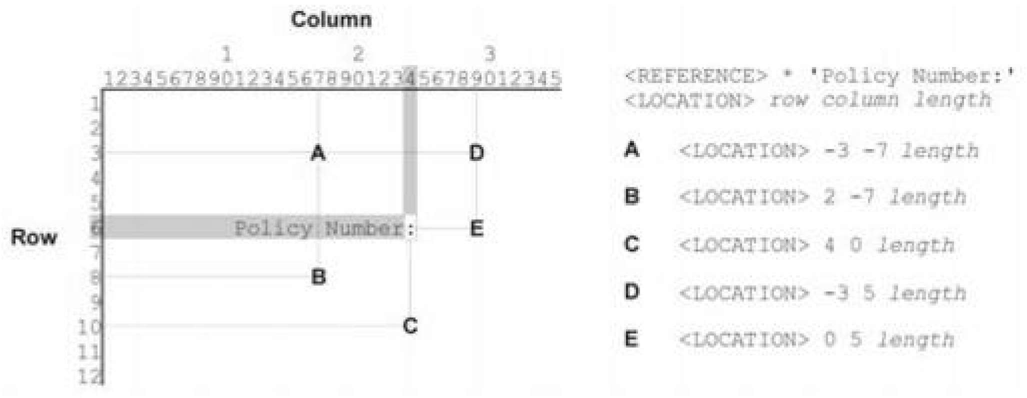
- A carriage control that identifies records on which to search for the reference

- Reference text for which to search
- The start column of the text

For complete information, see the *Enrichment Language Reference Guide*.

When you use a `<REFERENCE>` tag, Enrichment automatically sets the location of the field information relative to the reference point. As the following figure shows, this adjusts `<LOCATION>` tag syntax so that the row value defines the vertical offset of the field information relative to the reference line or record. Likewise, the column value defines the horizontal offset of the field information relative to the end of the reference point.

> **Note:** The `<LOCATION>` tag column value is always measured from the beginning of the last character of the reference point.

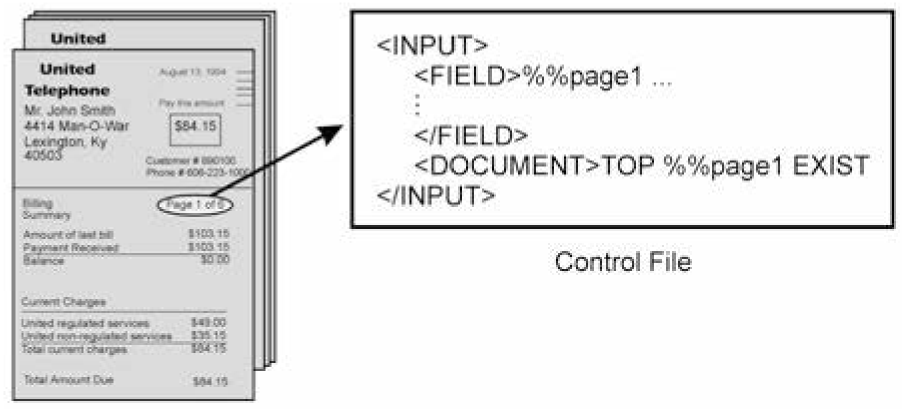


```
<REFERENCE> * 'Policy Number:'
<LOCATION> row column length

A   <LOCATION> -3 -7 length
B   <LOCATION> 2 -7 length
C   <LOCATION> 4 0 length
D   <LOCATION> -3 5 length
E   <LOCATION> 0 5 length
```

As this figure shows, set *row* to a negative number if the field information is the indicated number of lines or records above the reference record. Set *row* to 0 if the field information is on the reference record. Set *row* to a positive number if the field information is the indicated number of lines or records below the reference record.

Similarly, if you set the column parameter to a negative number, the field information begins that many characters before the beginning of the last character of the reference point, and so on.

For example, you might need to extract the account number from each bank statement you process. In the following figure, the location of the account number varies horizontally depending on the type of account and the number of transactions the customer has. The phrase `Account Number:` appears three spaces to the left of the account number, wherever it occurs on the page. The account number is always 12 characters in length. You could define a Field group as shown in the following figure to pick up the value.



```
<FIELD> %%Account_Number
    <REFERENCE> * 'Account Number:'
    <LOCATION> 0 3 12
</FIELD>
```

Since the account number does not occur with regularity on any specific record or line, the `<REFERENCE>` tag instructs Enrichment to search on all records for `Account Number:`. The account number's position can also vary from column to column, so no *start* parameter is specified on the `<REFERENCE>` tag.

The 0 after the `<REFERENCE>` tag indicates that the account number is on the same record as the phrase `Account Number:`. The 3 indicates that the account number begins on the third character position after the colon in `Account Number:`. Since the account number is 12 characters in length, the *length* parameter is set to 12.

Note: To set the value of the field to the reference string, set the `<REFERENCE>` tag as follows:

```
<LOCATION>0 -x y
```

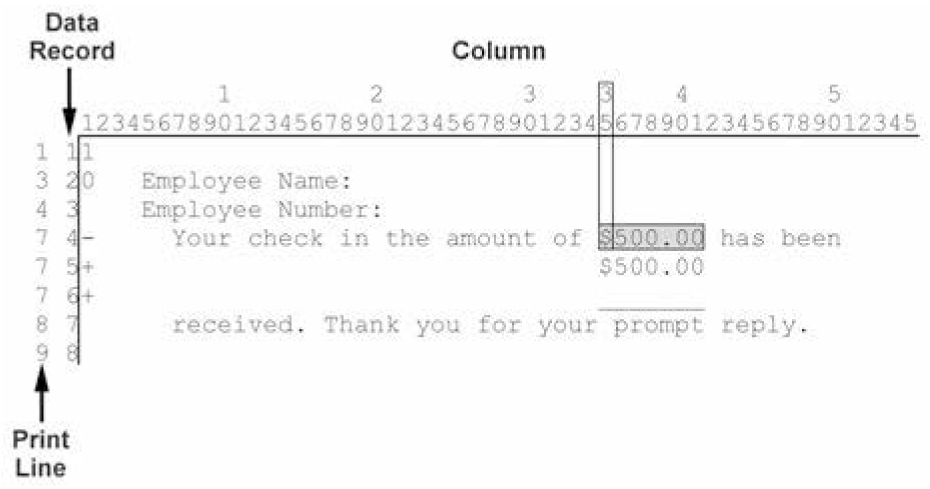
where x = length -1 and y = length.

Fields in Composed Print Streams

Metacode and AFPDS are “composed” print streams. They contain print control codes and other information that require different techniques to process using Enrichment. The Field tag group has several features to accommodate composed streams.

Finding Fields on Data Records Instead of Print Lines

You can set the Field group `<LOCATION>` tag's *method* parameter, which defines whether you specified the *row* value in print lines or data records, to *LINE* or *RECORD*. Print lines are the actual lines on which data prints on the page. Data records are the rows viewed when browsing or editing the data file online. The *method* parameter is only valid for ANSI or machine code line data. The default *method* value for line data (that is, for impact, AFP line, AFP mixed, and DJDE data) print streams is *LINE*. For AFPDS, Metacode, PCL and PostScript print streams, *method* is always *RECORD*.



Generally, you would only set the *method* parameter to *RECORD* for AFPDS or Metacode data. In fact, setting *method* to *RECORD* for line data only works if the record numbers stay the same for the entire input. Assuming that is so, you could use the tagging shown below to define a field whose value is the amount in the line data input shown in the diagram above.

```
<FIELD> %%Check_Amount
   <LOCATION> 4 35 7 RECORD
</FIELD>
```

Finding Fields on Particular Pages

You can restrict the pages on which Enrichment searches for a field by using the Field group `<ONPAGE>` tag. Refer to the *Enrichment Language Reference Guide* for further information.

You can specify a single page or a range of pages in the `<ONPAGE>` tag, depending on where you want the field information acted on. For example, to act on all occurrences of a field from page 2 to the next-to-last page of each document, you could set the `<ONPAGE>` tag to:

```
<ONPAGE> 2 L-1
or
<ONPAGE> M
```

Thus you can identify which pages to affect the field on without having to know the actual number of pages for each document and without each document having to have the same number of pages.

Extracting Fields Based on End Criteria

Typically, field information is extracted by picking up a specific number of characters (in impact, DJDE, AFP mixed, or AFP line print streams) or by picking up text until the *length* parameter value is met or an AFP code or Metacode occurs (in AFPDS, AFP mixed, or Metacode print streams). You may need to pick up data that contains AFP codes or Metacode commands or that occurs before a specific string, or you may need to pick up data until the end of a record. In such cases, AFP codes or Metacode commands that occur within the data to be picked up must be removed so the data can be used for sorting, address cleansing, and so on.

The Field group `<TEXTUNTIL>` tag enables you to pick up field information until a specific condition is met or until the `<LOCATION>` tag *length* parameter value is met. Each AFP code or Metacode within the extracted field information is treated as a single space in the text. `<TEXTUNTIL>` tag syntax is:

`<TEXTUNTIL>` *`type`* `[`*`value units`*`]`

The *type* parameter specifies when Enrichment should stop picking up field information (that is, when it should stop reading hexadecimal codes or Metacode commands as spaces, thus ending the field information). For information on the valid *type* parameter settings, see the *Enrichment Language Reference Guide*.

Note: Do not specify *value* if you set the `<TEXTUNTIL>` tag type *value* to *RECORD* or *TEXT*.

The *value* parameter defines a specific point at which Enrichment should stop picking up field information.

- If you set *type* to X, XR, Y, or YR, *value* specifies a size in units. If Enrichment encounters a move of this size or greater, it stops picking up field information.
- If you leave *value* blank for type X, XR, Y, or YR, Enrichment stops picking up field information at the first move encountered.
- If you set *type* to *STRING*, *value* specifies the string at which Enrichment should stop picking up field information. If the string contains spaces, you must enclose it in single or double quotation marks.

The *units* parameter defines the units in which the value you specified is set. Use the *units* parameter only if you set type to X, XR, Y, or YR.

When you use the `<TEXTUNTIL>` tag, Enrichment counts each hexadecimal code or Metacode in field information as a space, up to the limit specified in the tag or until the field information reaches the length specified in the `<LOCATION>` tag.

For example, to pick up the city, state, and ZIP Code™ information from the AFP mixed record shown below for address cleansing, you would specify `<TEXTUNTIL> X` in the Field group.

```
K0DLexington,0C01KY0C010F0405030C01K
KCB           4508  450E316     473AK
```

For this example, `<TEXTUNTIL> X` specifies that Enrichment should stop picking up data when it encounters an absolute x move in that data. Therefore, Enrichment stops picking up data at

`X'04C7031A'`. Enrichment reads intervening AFP codes (X'04C50018', X'04C5001E' and X'03F106') as spaces, so the field information is:

```
Lexington, KY 40503
```

Hexadecimal and Binary References

Generally, you use the `<REFERENCE>` tag to identify a reference point in Metacode in the same manner as you would for line data, with one important difference. Because Metacode controls are represented in hexadecimal format, you will probably need to specify references to Metacode commands and associated data in hexadecimal or binary format so Enrichment can locate it. For example, the Field tag group shown below tells Enrichment how to find the field information for an address line in Metacode data.

```
<field> %%Addr1 K
   <reference> x'01' x'8D0104A50B' 3
   <location> 0 43 30
</field>
```

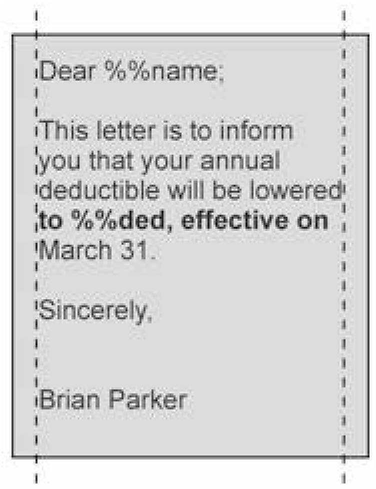
In the `<reference>` tag above, the *cc* parameter value is `x'01'`, indicating that Enrichment should look only on records with a carriage control of `x'01'` in column 1 for the reference point. The reference text for which Enrichment is to search is `x'8D0104A50B'`.

Replacing Fields and Substituting Inline Variables in Metacode Files

When Enrichment replaces fields in a Metacode input, the new value replaces the field differently than in line data. If you set the `<REPLACE>` tag *expandYN* parameter to *YES* and the field information begins in text, information to the right of the field moves left or right to accommodate the full length of the new field value. The text displaced by field replacement in a Metacode input does not wrap or reflow.

Similarly, when Enrichment substitutes an inline variable in a Metacode input, it inserts the value in place of the variable name. The displaced text does not wrap or reflow. Inline variables must normally be in ASCII format in Metacode data.

If, as shown in the diagram below, the field information or variable value is wider than the space occupied by the placeholder and text follows the placeholder on the same line, that text moves to the right. If replacing a field or substituting a variable forces other text on the same line past the right margin, such text continues into the right margin and possibly off the page.

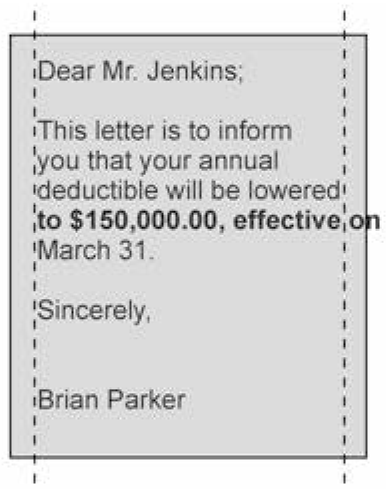


Dear %%name;

This letter is to inform you that your annual deductible will be lowered **to %%ded, effective on** March 31.

Sincerely,

Brian Parker



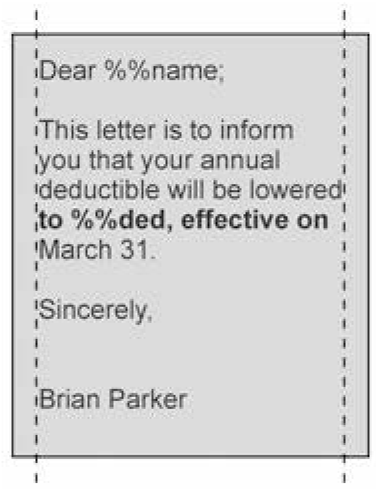
Dear Mr. Jenkins;

This letter is to inform you that your annual deductible will be lowered **to $150,000.00, effective on** March 31.

Sincerely,

Brian Parker

Conversely, if the field information or variable value is narrower than the space occupied by the placeholder and text follows the placeholder on the same line, the text moves to the left, as shown below.

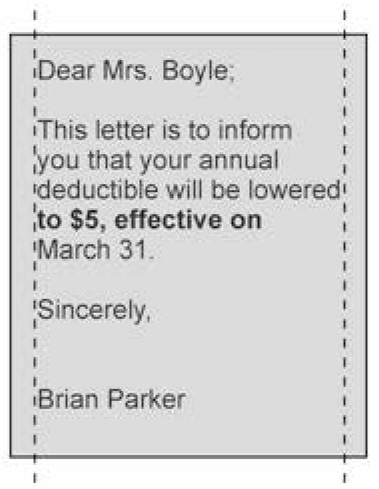


Dear %%name;

This letter is to inform you that your annual deductible will be lowered **to %%ded, effective on** March 31.

Sincerely,

Brian Parker



Dear Mrs. Boyle;

This letter is to inform you that your annual deductible will be lowered **to $5, effective on** March 31.

Sincerely,

Brian Parker

Fields in AFP Records

When working with fields in AFP records, note the following:

- When you extract a field from an AFP record, the field will end when a new AFP command is reached or the maximum length is reached (whichever comes first). So, for example, if you are picking up the text within a PTX record, the field will end when the next AFP command occurs or when the length of the field is reached.
- The length indicators on the AFP record will automatically be adjusted when a field replace is used to change the contents of a record.

- Normally, the text contents of PTX records are in EBCDIC (and are associated with an EBCDIC code page). When AFP streams are created on the PC or UNIX system, these records may contain ASCII data (because they are associated with ASCII code pages). You should use the `<TYPE>` tag *charset* parameter to specify this case.

Reading Keyed Information from Files

You can obtain keyed data from files (or VSAM on mainframe systems) by using the `LOOKUP` and `LOOKUPV` functions from the rule file. These high-level functions are designed to locate records within a file that correspond to a specified key.

Note: You can only use the LOOKUPV and WRITEV functions on mainframe systems.

To use `LOOKUP` and `LOOKUPV`, call them from the rule file when you want to obtain supplemental data. You need not open or close the files or be concerned with the actual mechanics of how Enrichment reads the file. When you call `LOOKUP` or `LOOKUPV`, the function returns the complete matching record. You should then parse out the data you need from the record.

These are key-based table lookups. You pass a key to the function, which looks in the file for a record with a matching key. If it finds the key, Enrichment returns the matching record.

Updating Keyed Data Files

The WRITEV and UPDATE functions write data to keyed files.

Note: If you want to use `WRITEV`, you must set the `LOOKUPV` function *access* argument to *W* (read/write).

Using the WRITEV Function to Update KSDS VSAM Files (Mainframe Only)

The `WRITEV` function updates existing VSAM files. You can use the `WRITEV` function to add, modify, or delete records. One record is processed for each call to `WRITEV`. You must ensure that the records are in the format expected by the VSAM file.

Enrichment can process any number of files simultaneously. As shown below, you can use `WRITEV` with Enrichment to update VSAM files that are used with the `LOOKUPV` function.

Note: When a sequential file is updated, it is written out in optimized form (sorted with blank records removed).

You cannot use the UPDATE function with tables.

```
/* Update the customer billed-to-date amount in the customer record */
%%custrec = LOOKUPV(DD:CUSTOMER,%%custnum) /* Get customer record   */
%%billed = RGET(%%custrec, I, 10)         /* Extract billed        */
                                           /*    amount            */
%%billed = %%billed + %%invoice_amount     /* Increase billed       */
                                           /*    amount            */
%%custrec = RPUT(%%custrec, %%billed, I, 10)/* Put back in record   */
%%result = WRITEV(DD:CUSTOMER, %%custrec, %%custnum)/* Update record */
```

Reading and Writing Record Data

The READ and WRITE functions read and write one record at a time from a file.

The READ function is normally used in the START: rules to initialize data or within the DOCUMENT: rules to read files associated with each document to get additional data. Refer to the *Enrichment Language Reference Guide* for more information on the READ function.

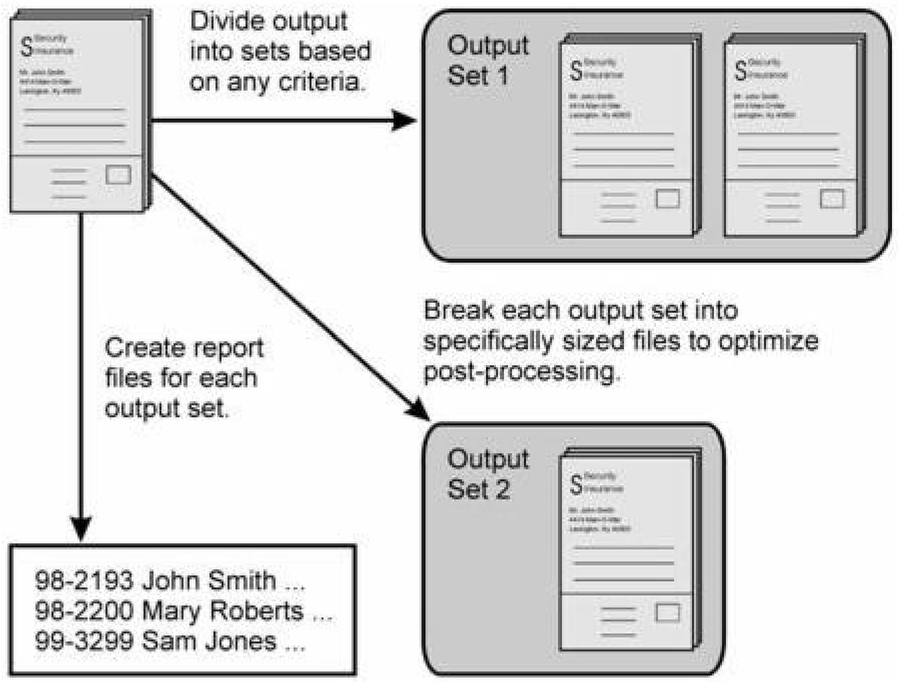
You can use WRITE to create reports, indexes or any other data on a sequential basis. Unlike the Sidefile tag group process, you can use WRITE to write multiple records per document or to only write records for some documents. The following illustrates the use of READ and WRITE in the rule file.

```
START:
  %%JOBNUM = READ(DD:JOBNUM)                <! Get job number.       >
  %%OUTPUTFILE = 'FILE:'| %%JOBNUM | '.RPT' <! Set output file name. >
  %%TOTALAMOUNT = 0
DOCUMENT:
  %%CUSTINFO = %%CUSTNUM | %%INVOICE
  %%TOTALAMOUNT = %%TOTALAMOUNT + %%INVOICE <! Add total invoices.   >
  WRITE(%%OUTPUTFILE,%%CUSTINFO)            <! Write customer record.>
FINISH:
  %%CUSTINFO = 'TOTAL:  ' | %%TOTALAMOUNT   <! Write total record.   >
  WRITE(%%OUTPUTFILE,%%CUSTINFO)
```

In the START: section, Enrichment reads the value of %%JOBNUM from a file and uses it to establish the name of the report file. In the DOCUMENT: section, Enrichment writes the invoice amount for each customer to the report. In the FINISH: section, Enrichment writes the total invoices to the report.

Defining Output

The Output tag group identifies parameters for creating one or more output print streams. You must place an Output tag group in the control file to identify each output from an Enrichment application. The Output group begins with the `<OUTPUT>` tag and ends with the `</OUTPUT>` tag. Neither tag has parameters. As the following figure illustrates, you can instruct Enrichment to break each output into multiple files so that no file exceeds a certain size. You can also specify any number of report files for each output file.


Divide output into sets based on any criteria.
Output Set 1
Break each output set into specifically sized files to optimize post-processing.
Create report files for each output set.
Output Set 2
98-2193 John Smith ...
98-2200 Mary Roberts ...
99-3299 Sam Jones ...

The Output tag group consists of several tags. Two of the most important are:

- `<FILE>` The path and file name for the output print stream. Each Output tag group can contain as many `<FILE>` tags as necessary to identify files to hold the output.
- `<NAME>` A unique reference name for the output print stream. You use this reference name to identify the output in other Enrichment tags. The `<NAME>` tag value cannot contain spaces. For example, if your output print stream will contain year-end dividend information, you might code the `<NAME>` tag as follows:

```
<NAME> YEAR_END_DIVIDENDS
```

For complete information on these and other tags in the Output tag group, see the *Enrichment Language Reference Guide*.

Sidefile Tag Group

Use the Sidefile tag group within the Output group to identify a flat file (called a “side file”) and data about each document in the output to write to the flat file. Side files are sometimes called data files or extract files. Each Sidefile tag group creates one data file that contains one record for each document processed in the specified output. You can use multiple Sidefile tag groups to create multiple side files. The Sidefile group begins with the `<SIDEFILE>` tag and ends with the `</SIDEFILE>` tag. Neither tag has parameters.

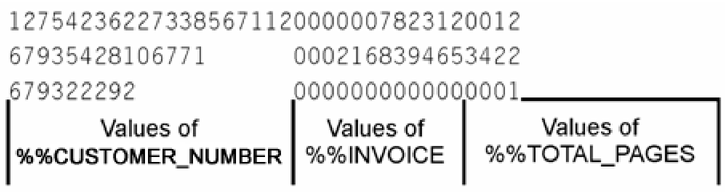
> **Note:** Using multiple `<SIDEPART>` tags allows for automatic record formatting. Alternately, you could use a single `<SIDEPART>` tag and format the record variable in the rule file.

The following figure shows a Sidefile tag group that creates an extract file that contains the customer number, invoice amount, and the total number of pages for each document.

```
<SIDEFILE>
   <FILE> C:\MyData\sidefile.txt
   <SIDEPART> %%CUSTOMER_NUMBER 20 L
   <SIDEPART> %%INVOICE 12 R 0
   <SIDEPART> %%TOTAL_PAGES 4 R 0
</SIDEFILE>
```

In this sample code, Enrichment writes the side file to `C:\MyData\sidefile.txt` as specified by the `<FILE>` tag. The customer number and invoice are user-defined variables, while *%%TOTAL_PAGES* is a system variable determined by Enrichment.

Assuming the output contains three documents, the extract file created by the example might resemble the data in figure below.



```
12754236227338567112000000782312001 2
67935428106771      0002168394653422
679322292           0000000000000001
```

In the above figure, the value of *%%CUSTOMER_NUMBER* in the first record is 20 characters in length, as defined in the `<SIDEPART>` tag. In each of the next two records, however, the *%%CUSTOMER_NUMBER* value is less than 20 characters in length, so it is left-justified and padded to the proper length with blanks. The *%%INVOICE* value has been right-justified and padded with zeroes in each record, as has the *%%TOTAL_PAGES* value.

Adding an AFP Index to an Output

An AFP Index is a group of records added to the top of each document that can be used with IBM's AFP Viewer, Large Mailing Operations (LMO) System, OnDemand, or other systems to identify the document and key information about it.

The layout and contents of the AFP Index are specified using the AFPIndex tag group within an Output tag group. The AFP Index groups all pages of a document within Begin Named Group (BNG) and End Named Group (ENG) structured fields. The contents of the AFP Index are the Tag Logical Element (TLE) structured fields placed at the top of each document.

You can add multiple TLE structured field records to each document by specifying multiple variables to include in the AFP Index. Each TLE record will contain:

- The title of the data item
- The system- or user-defined variable name

The AFPIndex tag group consists of three tags: `<AFPINDEX>` and `</AFPINDEX>`, which begin and end the tag group, and `<PART>`, which identifies a variable to include in the AFP Index. The `<AFPINDEX>` and `</AFPINDEX>` tags are required in the AFPIndex tag group, but they have no parameters.

The following shows a sample control file used to create an AFP Index that contains the customer name and page count for each document.

```
<input>
   <name> Bills                        <! Internal name for input file>
   <file> DD:INPUT1                    <! JCL name for input file     >
   <type> A                            <! Field to determine each     >
   <field> %%Customer_Name KA          <!  document and to index by   >
     <reference> ! 'Customer Name:'
     <location> 0 2 25
   </field>
   <doc> T %%Customer_Name C           <! New doc when name changes   >
</input>
<output>
   <name> IndexedOutput                <! Internal name for output    >
   <file> DD:OUTPUT1                   <! JCL name for output file    >
   <afpindex>                          <! Info to make into TLE record>
      <part> 'Customer Name:' %%Customer_Name 25<! User-defined val.>
      <part> 'Total Pages:' %%TOTAL_PAGES 8    <! System variable    >
   </afpindex>
</output>
```

Because you place the AFPIndex tag group within the Output tag group, the AFP Index only contains information about the documents written to a particular output.

Redirecting Input and Output at Run Time

UNIX and Windows users can redirect the input sources and output files by using pipes.

> **Note:** "|" is the pipe symbol. For additional information on redirection see your Windows or UNIX system documentation.

If the file parameter used in the `<INPUT>` group is `<FILE>|` the data will be read from the standard input (STDIN) that you specify when Enrichment is executed, such as the keyboard. On UNIX, standard input could also be a file or output from another program. To specify the input source, use the following syntax when you execute Enrichment:

- To use output from another application as the input (UNIX only),

  ```
  'sweaver < otherapplication'
  ```

- To use a file as input,

  ```
  sweaver < filename
  ```

- To use input from the keyboard,

  ```
  sweaver
  ```

If the filename parameter used in the `<OUTPUT>` group is `<FILE> |` the data will be written to the standard output (STDOUT) that you specify when Enrichment is executed, which is typically the display but could also be a file. To specify the output destination, use the following syntax when you execute Enrichment:

- To send the output to another application (UNIX only),

  ```
  'sweaver > otherapplication'
  ```

- To send the output to a file,

  ```
  sweaver > filename
  ```

- To send the output to the display,

  ```
  sweaver
  ```

Developing a Control File

A control file defines an application's input, output, and global actions (actions that you take on every document). It defines the overall print stream processing environment. Each Enrichment application requires one control file.

Key options configured in the control file include:

- Input file characteristics (such as type, field locations, and page and document break indicators)
- Rule file characteristics
- Added object size, content, and placement
- Information to control addition of pages
- Parameters for CASS Certified™ address cleansing
- Parameters for postal Presort processing
- Control information (such as supplemental data files, rule files, and inserts)
- Output file characteristics (such as name, separation requirements, and convenience break parameters)
- Performance options

Creating a Control File

To create a control file, follow this general process.

1. Using either Visual Engineer or a text editor, create a new file. (For instructions on creating a new control file in Visual Engineer, see the Visual Engineer online help.)
2. Define each input print stream.
   - Use a separate Input tag group for each input print stream.
   - Use Field tag groups to identify variable data to extract from documents.
   - Use the Input group `<DOCUMENT>` tag to specify how Enrichment identifies the top or bottom of a document.
   - If necessary, use the Input group `<PAGE>` tag to indicate how Enrichment finds new pages.
   - If you plan to CASS™ cleanse, presort, or locate the ZIP Code™ within an address, use the Address tag group or the `<ADDRESSBLOCK>` tag to identify the variables that make up the address components.
   - Code additional performance or layout attributes as necessary to fully define the input print stream.
3. Use the Insertpage and Insertrec tag groups to identify files (if any) that contain control records or electronic inserts to add.

4. Use Add tag groups to identify objects (barcodes, forms, images, or text) to add to every output. If the added objects vary based on the output, place the Add tag groups within the appropriate Output group, as discussed in step 8.
5. Use the Sortmatch tag group to identify sort and match criteria for sorting and consolidating documents.
6. Use the Rule tag group to identify the rule file for performing conditional processing, if any. For more information on rule files, see **Creating a Rule File** on page 81.
7. Use the CASS tag group in conjunction with the Input group `<CLEANSE>` tag to specify postal address cleansing requirements.
8. Use Output tag groups to identify the output files to create:
   - Specify multiple file names if you do not want the individual files to exceed a certain size (in bytes, lines, pages, documents, or trays) or use the `<DYNAFILE>` tag to automatically name files that are dynamically allocated.
   - If you want to conduct a presort, use the Presort tag group to indicate the method.
   - Use the Sidefile tag group to indicate the names and contents of any extract files and Reprint Indexes to create.
   - Use the Output group `<FILEMAX>` tag to indicate how to automatically break output into convenient groups of documents.
   - Use the AFPindex tag group to indicate Tag Logical Elements (TLEs) before each document for indexing full AFPDS data.

Control File Tags

Enrichment contains the following major tag groups. There may be sub-tag groups within the major tag groups for use in the control file. For more information, see the *Enrichment Language Reference Guide*.

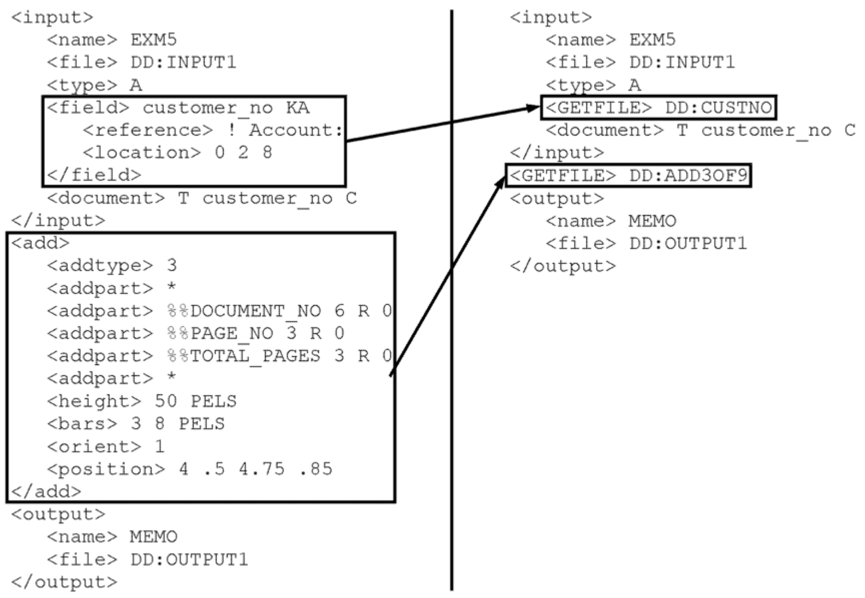
Input Tag Group	Defines a print stream to process and the parameters used to process it.
Add Tag Group	Identifies objects (text, graphics, forms, or barcodes) to print on outputs.
Sortmatch Tag Group	Identifies one or more inputs to sort or match by fields.
CASS Tag Group	Defines the CASS Certified™ program and parameters for cleansing addresses specified using the Address tag group. This tag group is also used for adding delivery point barcodes.
Insertpage Tag Group	Identifies a file that contains one or more pages to insert into a document where a specified variable name exists.
Insertrec Tag Group	Identifies a file that contains one or more records to insert into a document where a specified variable name or field exists.
Rule Tag Group	Defines a file that contains conditions used by Enrichment in rule-based (that is, conditional) document assembly.

Output Tag Group	Defines one or more print streams for Enrichment to create.
Environment Tag Group	Tunes Enrichment processing performance and sets environmental parameters.
Banner Tag Group	Identifies one or more pages to use as a banner.
Table Tag Group	Identifies an alternate content source for the LOOKUP function.

Sharing Code Between Applications

As you develop Enrichment applications, you may find that specific tag groups in your control files contain the same information from one application to the next. In this case you can use the `<GETFILE>` statement to call files that contain the repeated information.

The following diagrams illustrate how you can use `<GETFILE>` to share code that defines a field and adds a 3of9 barcode.



```
<input>
   <name> EXM5
   <file> DD:INPUT1
   <type> A
   <field> customer_no KA
      <reference> ! Account:
      <location> 0 2 8
   </field>
   <document> T customer_no C
</input>
<add>
   <addtype> 3
   <addpart> *
   <addpart> %%DOCUMENT_NO 6 R 0
   <addpart> %%PAGE_NO 3 R 0
   <addpart> %%TOTAL_PAGES 3 R 0
   <addpart> *
   <height> 50 PELS
   <bars> 3 8 PELS
   <orient> 1
   <position> 4 .5 4.75 .85
</add>
<output>
   <name> MEMO
   <file> DD:OUTPUT1
</output>
```

```
<input>
   <name> EXM5
   <file> DD:INPUT1
   <type> A
   <GETFILE> DD:CUSTNO
   <document> T customer_no C
</input>
<GETFILE> DD:ADD3OF9
<output>
   <name> MEMO
   <file> DD:OUTPUT1
</output>
```

For more information, see the *Enrichment Language Reference Guide*.

Securing a Control File

You can convert a control file to an unreadable format for security purposes by compiling the control file. You can decompile a control file at any time to convert it back to a readable format. Any rule file that is contained in-line in the control file is also compiled and decompiled. Compiling a control file only changes the format of the data in the control file so that it cannot be read. It does not change the content of the control file or the way Enrichment operates.

To compile a control file, specify the run-time arguments explained in the table below in the Enrichment JCL.

Table 6: Run-Time Arguments for Compiling and Securing a Control File

Argument	Description
F=filename	Specifies the name of the file or DD in which the compiled or decompiled control file is stored. The default compile DD is DD:COMPCNTL. The default decompile DD is DD:DECNTL.
K=key	Specifies a key used to compile or decompile control files. The key can be up to 10 characters and is used to lock or unlock the control file. Any rule included within the control file with the Content tag group is also compiled or decompiled using the key.

The first time a control file is processed using a specific key, it is compiled. The next time it is run using the same key, it is decompiled. Enrichment can run from the original, compiled, or decompiled control file even if you do not specify a key.

Sample Compile JCL

The following is sample JCL for compiling a control file.

```
//JOBCARD
//JOBLIB, DD DSN=STREAM.WEAVER.LOAD,DISP=SHR
//        DD DSN=SYS3.CRUNTIME.LOAD,DISP=SHR
//*-------------------------------------------------------------
//* The first time you run an Enrichment job with the K
//* parameter, it places a compiled version of the control file
//* in the COMPCNTL DD.
//*-------------------------------------------------------------
//COMPILE  EXEC PGM=PDRSW000,PARM='//K=COMPIT',REGION=0M
//REPORT   DD SYSOUT=*
//CONTROL  DD DSN=STREAM.WEAVER.CONTROL(UNCOMPED),DISP=SHR
//COMPCNTL DD DSN=STREAM.WEAVER.COMPCNTL(COMPILED),DISP=SHR
//INPUT1   DD DSN=STREAM.WEAVER.INPUT,DISP=SHR
//OUTPUT1  DD DSN=STREAM.WEAVER.OUTPUT1,DISP=SHR
//*-------------------------------------------------------------
```

Sample Decompile JCL

The following is a sample JCL for decompiling a control file.

```
//JOBCARD
//JOBLIB   DD DSN=STREAM.WEAVER.LOAD,DISP=SHR
```

```
//         DD DSN=SYS3.CRUNTIME.LOAD,DISP=SHR
//*-----------------------------------------------------------------
//* The first time you run an Enrichment job with the K
//* parameter, it places a compiled version of the control file
//* in the COMPCNTL DD.
//*-----------------------------------------------------------------
//COMPILE  EXEC PGM=PDRSW000,PARM='//K=COMPIT',REGION=0M
//REPORT   DD SYSOUT=*
//COMPCNTL DD DSN=STREAM.WEAVER.CONTROL(UNCOMPED),DISP=SHR
//CONTROL  DD DSN=STREAM.WEAVER.COMPCNTL(COMPILED),DISP=SHR
//INPUT1   DD DSN=STREAM.WEAVER.INPUT,DISP=SHR
//OUTPUT1  DD DSN=STREAM.WEAVER.OUTPUT1,DISP=SHR
//*-----------------------------------------------------------------
```

Sample JCL for Running Compile or Decompile

The following is a sample JCL for running Enrichment with a compiled control file.

```
//JOBCARD
//JOBLIB   DD DSN=STREAM.WEAVER.LOAD,DISP=SHR
//         DD DSN=SYS3.CRUNTIME.LOAD,DISP=SHR
//*-----------------------------------------------------------------
//* Allow users access to the compiled control file and they
//* can run Enrichment normally (without a key) using the
//* compiled control file in their CONTROL DD.
//*-----------------------------------------------------------------
//RUNIT    EXEC PGM=PDRSW000,REGION=0M
//REPORT   DD SYSOUT=*
//CONTROL  DD DSN=STREAM.WEAVER.COMPCNTL(COMPILED),DISP=SHR
//INPUT1   DD DSN=STREAM.WEAVER.INPUT,DISP=SHR
//OUTPUT1  DD DSN=STREAM.WEAVER.OUTPUT1,DISP=SHR
//* ---------------------------------------------------------------
```

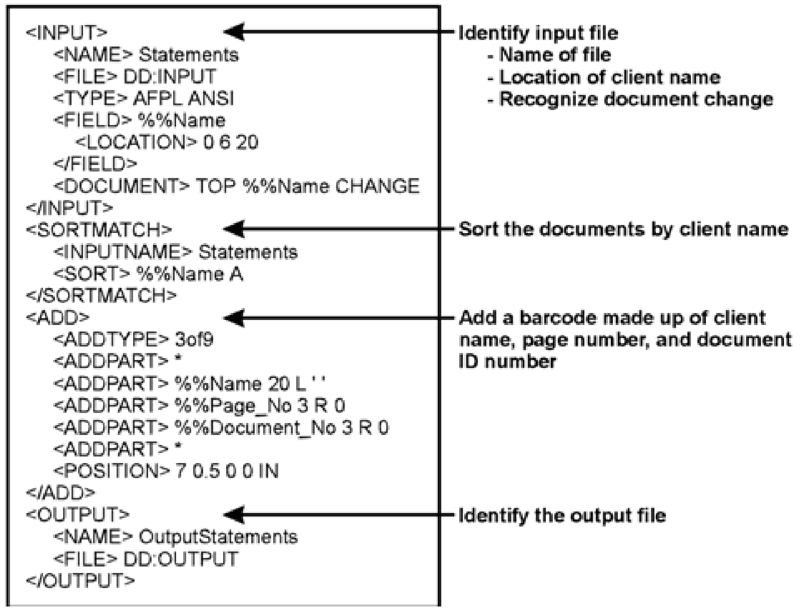
Example Control File

The following example of a simple control file contains one input, one output, and processing instructions to sort and add a 3of9 barcode.

The example shows tag groups that describe specific objects (such as inputs) or processes. Within the tag group are tags that are used to more fully define the object. Parameters may be required to fully define each tag.

Note: When a tag includes multiple parameters, you must separate each parameter value from the next with one or more spaces. Furthermore, you must place parameters in the proper syntax order.

In the example, the `<INPUT>` tag begins a tag group (the `</INPUT>` tag ends the group). The `<NAME>` tag describes the Input tag group, and `Statements` is the value of the `<NAME>` tag parameter. This is the name that is used to refer to this input in other areas of the control file.



```
<INPUT>
   <NAME> Statements
   <FILE> DD:INPUT
   <TYPE> AFPL ANSI
   <FIELD> %%Name
      <LOCATION> 0 6 20
   </FIELD>
   <DOCUMENT> TOP %%Name CHANGE
</INPUT>
<SORTMATCH>
   <INPUTNAME> Statements
   <SORT> %%Name A
</SORTMATCH>
<ADD>
   <ADDTYPE> 3of9
   <ADDPART> *
   <ADDPART> %%Name 20 L ' '
   <ADDPART> %%Page_No 3 R 0
   <ADDPART> %%Document_No 3 R 0
   <ADDPART> *
   <POSITION> 7 0.5 0 0 IN
</ADD>
<OUTPUT>
   <NAME> OutputStatements
   <FILE> DD:OUTPUT
</OUTPUT>
```

Identify input file
- Name of file
- Location of client name
- Recognize document change
Sort the documents by client name
Add a barcode made up of client name, page number, and document ID number
Identify the output file

While it may be easier to read a control file in which each tag occupies its own line (as shown in the example), you can also enter the data without the line breaks, as shown below.

```
<input><name>MONTHLY<file>DD:SYSIN<type>AFPDS</input>
```

Developing a Rule File

A rule file defines conditional processing. It is an optional set of code that offers additional control and user processing for individual documents and pages. In the rule file you can change field values based on certain conditions, perform calculations, create and use variables, route documents to outputs, read and write files, and add inserts or banner pages. The rule file is compiled as part of control file processing.

While the rule file can be a separate file, developers often embed rule file code within the control file. When it is embedded in this manner, there is no separate file that contains the rule file code. Nevertheless, rule file code that is located in the control file is still referred to as the rule file.

Creating a Rule File

To create a rule file, follow this general process. For more information about any of the steps, see the *Enrichment Language Reference Guide*.

1. Decide whether you want to embed the rule file in the control file or whether you want to create a separate rule file that you call from the control file. Most often developers choose to embed the rule file in the control file.
2. Add the Rule tag group to the control file.
3. Do one of the following:

   - If you are embedding the rule file in the control file, add the Content tag group to the Rule tag group. The Content tag group will contain the rule file you write.

     An example of the Content tag group is shown below.

```
<rule>
<content>
rule file statements
</content>
</rule>
```

   - If you are calling an external rule file, add the <FILE> tag to the Rule tag group and specify the path to the external rule file.

```
<rule>
<file>c:\rules\myRuleFile.txt
</rule>
```

4. In the rule file (either the external or embedded rule file) create the sections required for your application. Your rule file can contain some or all of the sections listed below. Each section executes at a different time during Enrichment processing. The sections consist of a section header (the section name followed by a colon) and one or more statements. A section header is the section name followed by a colon.

   For example:

```
DOCUMENT:
 %%SUM = %%A + %%B
```

   Note: If you do not designate a section name, Enrichment assumes the section is `DOCUMENT:`.

   Your rule file can contain some or all of the following sections.

START	Executes before any of the inputs are read. Use this section to initialize variables and add beginning banner pages. You should also declare user-written functions and set the initial file name for dynamically allocated outputs in this section.
DOCUMENT	Executes once for each document as soon as the document has been identified and optionally matched with other inputs. You can use this section to route the document to one or more outputs, append inserts, and add banner pages. When inputs are sorted using `<SORTMATCH>` or `<DOUBLESORT>`, documents are processed in the sort order, not in the order in which they appear in the original input. This section was the rule file in versions of Enrichment prior to 4.x.
PRESORTED	Executes once for each document after postal Presort has occurred. You can add banner pages and reroute the document to a different output (for example, to outsort residual mail pieces). This section only executes on documents routed to an output containing the Presort group. The PRESORTED section processes every document for which a record is returned from the external program called by the `<PRESORT> <STEP>` tags. Documents that are not returned by the external program are instead routed to a reject file if the tag `<REJECTFILE>` is specified in the Presort tag group. If the tag `<REJECTFILE>` is not specified, documents that are not returned may be "lost," meaning that they will not appear in any Enrichment output.
PAGE	Executes once for each page in a document immediately prior to writing the page to the output. You can use this section to apply page-varying information, such as checksum for an inserter barcode. This section was the pagerule in versions of Enrichment prior to 4.X.
FINISH	Executes once after the last document is processed. You can compute summary information and add banner pages to any or all outputs.

5. In each section you create, write the appropriate code to accomplish what you want with your application. The code that you write may incorporate the following items.

User-Defined Variables	These let you change fields, insert page file assignments, and insert record file assignments with Set Statements and functions.
User-Written Functions	You can use existing subroutines within rule files as functions or write your own. These subroutines can be written in COBOL, Assembler, or C. For more information, see **Working with User-Written Functions** on page 88.

Functions	There are a rich set of functions for you to use in rule files. You can use these functions to manipulate data and in some cases, return the result as a variable value.
Instructions	These allow you to perform conditional processing. Instructions include IF THEN ELSE, QUIT, FOR…NEXT, DO…NEXT, and SELECT CASE.
Print Stream Commands	These let you manipulate your documents from within the rule file. Print stream commands include <OUTPUT>, <APPEND>, <BANNER>, and <FILEBREAK>.
Comments	These annotate the rule file and can be placed anywhere within it.

User-Defined Variables

You can assign variables in the rule file or at the command line using the U switch. For more information about assigning variables at the command line, see **Running an Application** on page 200.

Variable assignments have the following form:

```
%%variable = expression
```

The simplest expression is just to reference another variable. For example:

```
%%barcode2 = %%barcode1
```

Enrichment supports a large variety of more complicated expressions. These fall into three groups: string expressions, numeric expressions, and logical expressions.

Note: Numeric values in Enrichment cannot be more than 9 digits long. If you need to handle longer numbers, you can use the SUBSTR function to break up the number into smaller workable parts.

String Expressions

String expressions result in the creation of a character string. String expressions include:

- String constants. If a string constant contains any spaces or symbols, it must be delimited by single or double quotes. String constants can also be ASCII, EBCDIC, or hexadecimal.
- Two or more expressions combined with the concatenation operator "|". For example:

```
%%x = %%y | %%z | string
```

- A string function. For example:

```
%%x = SUBSTR(%%y, 2, 10)
```

Numeric Expressions

Numeric expressions result in an integer value. Numeric expressions include:

- Numeric constants. These can also be in binary form. For example:

  ```
  %%x = 23
  ```

  ```
  %%x = b'00101101'
  ```

- Numeric expressions combined with arithmetic operators. Most of these operators take two operands, such as %%y + %%x. However, the unary minus operator has only one operand. For example:

  ```
  %%x = -%%y
  ```

- A numeric function. For example:

  ```
  %%x = LENGTH(%%y)
  ```

> **Note:** Numeric values in Enrichment cannot be more than 9 digits long. If you need to handle longer numbers you can use the SUBSTR function to break up the number into smaller parts that you can then work with.

Arrays

An array is an indexed list of values. For example, if you have an array named `%%myArray` with values 23, 12, and 65, you could reference these values as follows:

```
%%myArray[0] would return 23
%%myArray[1] would return 12
%%myArray[2] would return 65
```

This array has a size of three meaning that there are three values in the array. The size of an array is specified when it is declared which is accomplished through the `DECLARE` function. For more information on this function, see the *Enrichment Language Reference Guide*.

When you use an array in an assignment, certain rules apply:

- If you use an array on the left side of an assignment, you will set the value of all elements of the array. For example:

  ```
  %%myArray = 33
  ```

  will set all elements of the array to 33.

  However, the following:

  ```
  %%myArray[2] = 33
  ```

  will set the 3rd element of the array to 33.

- You cannot use an array on the right side of an assignment unless you use the `DECLARE` or `ARRAYSIZE` functions. For example, the following statements are valid:

  ```
  %%someValue = DECLARE(%%myarray, 'A', 3)
  ```

  ```
  %%anotherValue = ARRAYSIZE(%%myArray)
  ```

Logical Expressions

Logical expressions result in the value TRUE or FALSE. These are represented as integers where 0 is FALSE and anything else (usually 1) is TRUE. Logical expressions are normally used within IF THEN ELSE instructions and DO loops. They include:

- Expressions combined with the comparison operators. For example:

  ```
  %%x = %%y > %%z
  ```

  (%%x will be 1 (TRUE) if %%y is greater than %%z.)

  ```
  %%x = %%y = %%z
  ```

  (%%x will be 1 (TRUE) if %%y equals %%z)

- Expressions combined with the logical connectors. The AND and OR connectors combine two expressions, while the NOT connector negates the single expression following it. For example:

  ```
  %%x = %%y AND %%z
  ```

  (%%x will be 1 (TRUE) if both %%y and %%z are TRUE (not 0)).

- A logical function. For example:

  ```
  %%x = CHANGED(%%y)
  ```

  (%%x will be either 1 (TRUE) or 0 (FALSE)).

Enrichment automatically converts string expressions to numeric expressions and vice versa. Therefore, you can use any field variable as a number or use numeric calculations as strings.

Assigning Statements

You can use rules to assign values to variables through:

- Concatenation
- Math operations
- Function calls

Concatenation

You can use a concatenation character (|) as follows to combine two or more strings or variables into a single variable (%%varname):

```
%%varname = value1 | value2 | value3 |…
```

For example, if the value of %%invoice is $21.00, then

```
%%MSG = 'Total amount due: ' | %%invoice | '.'
```

would result in `%%MSG` being equal to "Total amount due: $21.00".

> **Note:** On mainframe systems you can use either a solid vertical bar (`|, EBCDIC X’4F’`) or a broken vertical bar (`¦, EBCDIC X’6A’`) as a concatenation character. On UNIX systems, the vertical bar is sometimes displayed solid, sometimes broken, but is always `ASCII X’7C’`.

Math Operations

The table below lists and describes valid Enrichment math operators. Math operations are processed before concatenation. Enrichment evaluates these expressions from left to right according to the indicated operator precedence. If a mathematical expression contains parentheses, Enrichment evaluates expressions within the parentheses first.

Table 7: Enrichment Math Operators

Operator	Description	Precedence
+	Addition	3
-	Subtraction	3
*	Multiplication	2
%	Integer Division—returns the integer result when one value is divided by another **Note:** When using integer division of variables in a rule file, at least one space must separate the % math operator from the variables. For example, `%%var1%%%var2` is incorrect, but `%%var1 %%%var2` (with space after `%%var1`) is correct.	2
#	Remainder—returns only the remainder when one value is divided by another	2
(and)	Open and close parentheses	1

You can nest parentheses as deeply as necessary, as long as every open parenthesis has a complementary closing parenthesis. The table below shows how the results of Enrichment math expressions can differ based on the evaluation order.

Table 8: Example of Evaluation Order

Expression	Result
(2 + 5) * 10 - 4 % 2	68
(2 + 5 * 10) - 4 % 2	50
2 + 5 * 10 - 4 % 2	50
2 + 5 * (10 - 4 % 2)	42
2 + (5 * 10 - 4) % 2	25
(2 + 5) * (10 - 4) % 2	21
2 + (5 * (10 - 4)) % 2	17
(2 + (5 * (10 - 4))) % 2	16

Function Calls

Enrichment includes a wide range of functions that you can use in the rule file for string manipulation, conversion, formatting, and file I/O.

Note: You can also write your own functions. For more information, see **Working with User-Written Functions** on page 88.

Setting a Counter

Setting a counter is a simple two-part process accomplished in the rules:

1. Initialize a variable

   In the `START:` section of the rules, name a variable to be used as the counter and set it to 0 as follows:

   ```
   %%Total_Subscribers = 0
   ```

2. Increment the variable

   In the `DOCUMENT:` or `PAGE:` section of the rules, set the counter as follows:

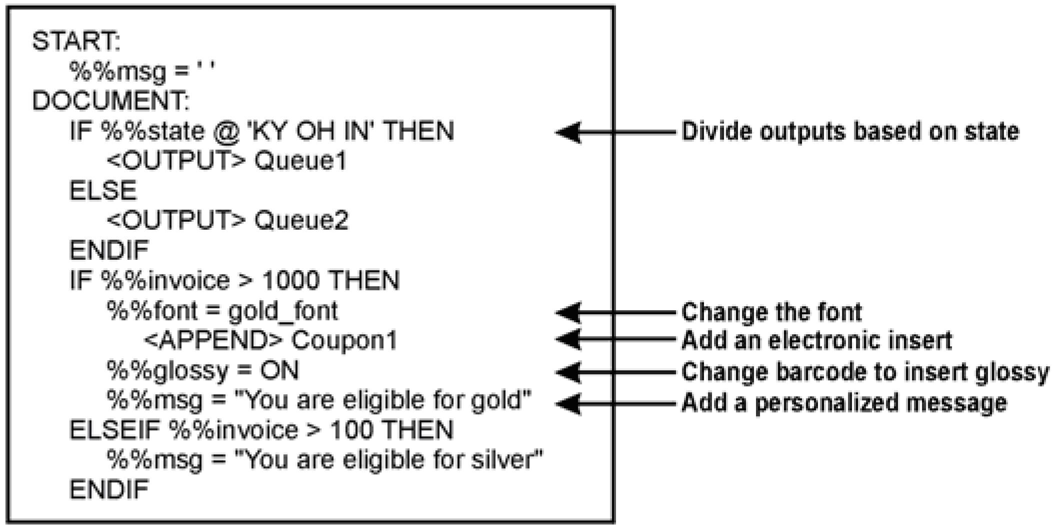
   ```
   %%Total_Subscribers = %%Total_Subscribers + 1
   ```

The following figure shows a counter that is set whenever the subscriber name changes.

```
START:
   %%Total_Subscribers = 0
DOCUMENT:
   IF CHANGED(%%Subscriber_Name) THEN
      %%Total_Subscribers = %%Total_Subscribers + 1
   ENDIF
```

Example Rule File

The example below shows a rule file that divides outputs based on the value of one variable and changes the appearance and contents of the outputs based on the value of another variable.



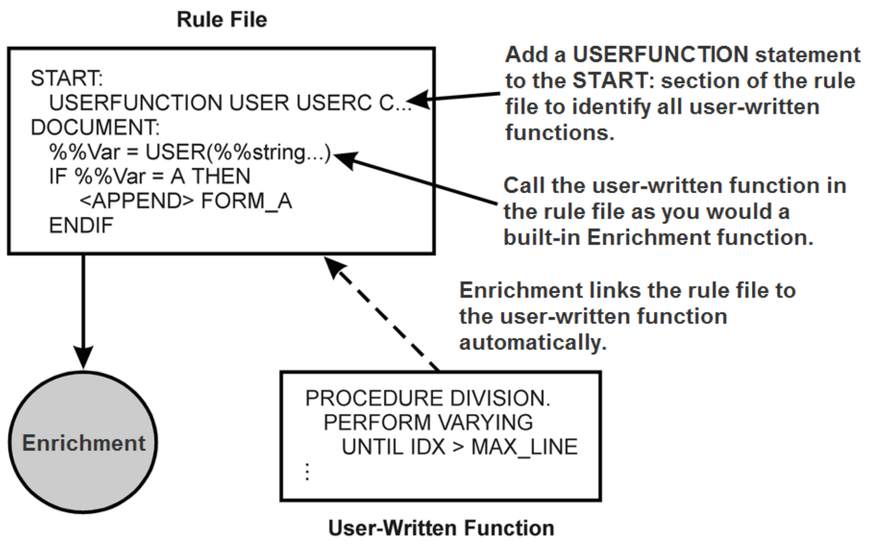
```
START:
   %%msg = ' '
DOCUMENT:
   IF %%state @ 'KY OH IN' THEN
      <OUTPUT> Queue1
   ELSE
      <OUTPUT> Queue2
   ENDIF
   IF %%invoice > 1000 THEN
      %%font = gold_font
         <APPEND> Coupon1
      %%glossy = ON
      %%msg = "You are eligible for gold"
   ELSEIF %%invoice > 100 THEN
      %%msg = "You are eligible for silver"
   ENDIF
```

Divide outputs based on state
Change the font
Add an electronic insert
Change barcode to insert glossy
Add a personalized message

Working with User-Written Functions

In addition to the functions built into Enrichment, you can write or use existing subroutines within rule files as functions. These functions can be written in COBOL, Assembler, or C. You identify the user-written subroutine for use within the rule file by adding the USERFUNCTION declaration in that file. Once you define a function, you can use it in the same manner as any other Enrichment function.

The following diagram illustrates how the interface for the user-written function works.


Rule File
START:
USERFUNCTION USER USERC C...
DOCUMENT:
%%Var = USER(%%string...)
IF %%Var = A THEN
<APPEND> FORM_A
ENDIF
Add a USERFUNCTION statement to the START: section of the rule file to identify all user-written functions.
Call the user-written function in the rule file as you would a built-in Enrichment function.
Enrichment links the rule file to the user-written function automatically.
Enrichment
PROCEDURE DIVISION.
PERFORM VARYING
UNTIL IDX > MAX_LINE
User-Written Function

The rules shown below define a user-written function called `ims_database`—COBOL load module `FNIMS2`—that uses an account number to look up a client's age in an IMS database. Either field `%%Account` or `%%Account2` can contain the customer account. `FNIMS2` is a normal user-written function subroutine with an input string of up to 25 bytes that returns an output string (`%%Age`) of up to 3 bytes. The Input call area (`maxin`) is 65 bytes (25+40) and the Output call area (`maxout`) is 43 bytes (3+40).

```
USERFUNCTION ims_database FNIMS2 COBOL N 65 43
IF %%Account <> ' ' THEN
   %%Age = ims_database(%%Account)
ELSE
   %%Age = ims_database(%%Account2)
ENDIF
```

The `USERFUNCTION` keyword identifies the function within the rule file. Once identified, Enrichment can access the function using the same syntax as with built-in functions, with one exception: you can pass only one variable to and from the user-written function.

To pass or receive more than one variable, concatenate multiple arguments into one and use the built-in `RGET` or `SUBSTR` functions to split data into multiple variables for the function's return value.

Using User-Written Functions

To use a user-written function, follow these steps.

1. Write the function. See **Writing a User-Written Function** on page 90.

2. Compile and link the function. See **Compiling and Linking User-Written Functions** on page 97.
3. Declare the function. See **Declaring User-Written Functions** on page 102.
4. Call the function where needed. See **Calling User-Written Functions** on page 103.

Writing a User-Written Function

The language you choose to create a function depends on the platform on which you are running Enrichment.

Mainframe	COBOL, Assembler, C If you are running one of the Enrichment load modules compiled for IBM (Language Environment or C/370), we recommend putting the run time library and COBOL in the link pack area (LPA) to improve performance. This is especially critical if you will be using a COBOL function with Enrichment. This move will increase processing speed because mainframe systems switch modes and reload faster when reading from memory than from disk.
UNIX	C
Windows	Any language that creates DLLs and uses CDECL linkage.

Note: Visual Basic 6.0 and earlier creates DLLs only with STDCALL linkage. These DLLs will not work with Enrichment.

When naming user-written functions, keep in mind that user-written functions cannot have the same names as Enrichment built-in functions.

Example COBOL User-Written Function

The following illustrates a COBOL user-written function.

```
/* ---------------------------------------------------------------- */
/* Rule File that calls COBOL user-written function                 */
/* ---------------------------------------------------------------- */
/* Cobol Function to reverse Characters                             */
/* ---------------------------------------------------------------- */
UserFunction REV_COB NORMCOB
/* Invoke User Functions */
%%Rev_Name_Cob_Normal = REV_COB(%%Whole_Name)
```

The name `REV_COB` refers to the COBOL function `NORMCOB` as defined in the `USERFUNCTION` command. `NORMCOB` (shown below) is a user-written function that reverses an input string. It is a compiled function that is linked into Enrichment and called within the rule file. If `NORMCOB` required two variables from the print stream instead of one as in the example, an extra step would be necessary to concatenate the two variables together and pass them as one variable to the function. The separation of the values occurs within the user function. If NORMCOB returned two or more variables, all values would have to be concatenated before the return. The built-in functions SUBSTR or RGET can separate the return values into variables.

```
*****************************************************************
* File: NORMCOB                                                 *
* System: Enrichment                                  *
* Version: 6.6.2                                                *
* Language: COBOL (mainframe only)                              *
* Copyright (c)1993-2013 Precisely                    *
*---------------------------------------------------------------*
* Purpose: Sample COBOL User-Written Function for "Normal"      *
*          type Enrichment User-Written Function interface.   *
*          This example reverses the input string.              *
*****************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. NORMCOB.
*
*****************************************************************
 ENVIRONMENT DIVISION.                                          *
*****************************************************************
DATA DIVISION.
*----------------------------------------------------------------
WORKING-STORAGE SECTION.
01      IDX         PIC 999 COMP.
01      IDX2        PIC 999 COMP.
*
*****************************************************************
*                    LINKAGE SECTION                            *
*---------------------------------------------------------------*
*         Declarations for Input/Output call areas to pass data *
*         between the Rule file and this function.              *
*****************************************************************
LINKAGE SECTION.
*--- Input Call Area -------------------------------------------
01 INPUT-CALL-AREA.
*    --- Required fields:
      05 IN-SIGNATURE PIC X(4).
      05 CALL-TYPE PIC X(1).
      05 CALL-FROM PIC X(1).
      05 FILLER PIC X(2).
      05 IN-RC PIC S9(9) COMP.
      05 IN-RV PIC S9(9) COMP.
      05 FILLER PIC X(20).
      05 IN-SIZE PIC S9(9) COMP.
*    --- User defined PIC X fields from Rule file arguments:
      05 IN-DATA PIC X(40).*
*--- Output Call Area ------------------------------------------
01 OUTPUT-CALL-AREA.
*    --- Required fields:
      05 OUT-SIGNATURE PIC X(4).
      05 OUT-RC PIC S9(9) COMP.
      05 OUT-RV PIC S9(9) COMP.
      05 FILLER PIC X(24).
      05 OUT-SIZE PIC S9(9) COMP.
*    --- User defined PIC X fields for combined results:
      05 OUT-DATA PIC X(40).
*
*****************************************************************
PROCEDURE DIVISION USING INPUT-CALL-AREA OUTPUT-CALL-AREA.
*
FUNCTION-START.
*
*   --- Check Input and Output signatures ---------------------
      IF IN-SIGNATURE NOT EQUAL 'PDRI' THEN
         MOVE -3 TO OUT-RC
         GO TO FUNCTION-END
      END-IF.
      IF OUT-SIGNATURE NOT EQUAL 'PDRO' THEN
         MOVE -3 TO OUT-RC
         GO TO FUNCTION-END
      END-IF.
*
*   --- Perform User function HERE: ---------------------------
*       -- Print some stuff
*       -- Reverse string
      INITIALIZE OUT-DATA, IDX2.
      PERFORM VARYING IDX FROM IN-SIZE BY -1
         UNTIL IDX EQUAL 0
         ADD 1 TO IDX2
         MOVE IN-DATA (IDX:1) TO OUT-DATA (IDX2:1)
      END-PERFORM.
*
*   --- Store return value, size and set return code & value --
   MOVE IN-SIZE TO OUT-SIZE.
   MOVE 0 TO OUT-RC.
   MOVE 0 TO OUT-RV.
*
 FUNCTION-END.
 EXIT.
*** End of NORMCOB COBOL II File **************************
```

Example C User-Written Function

The following illustrates a C user-written function that reverses an input string. The example below is for a User Function to be run on a UNIX system.

```
/*********************************************************************/
/* File   : normc.c                                                */
/* System : Enrichment                              */
/* Version: 6.6.2                                                  */
/* Copyright (c)1993-2013 Precisely                */
/* all rights reserved.  Unauthorized use or duplication prohibited. */
/*-----------------------------------------------------------------*/
/* Purpose:                                                         */
/*-----------------------------------------------------------------*/
/* History:                                                         */
/* static char SccsID[] = @(#)normc.c    1.1 08/18/97               */
/* 07/09/98 rolson Initial 50                                       */
/* 08/18/97 tcampbel Userfunction for testcases                     */
/* ------------------------------------------------------------------ */
/*
 * normc.c - Example Normal C User Function
 * Reverse characters.
 */
/* ------------------------------------------------------------------ */
#include <stdio.h>
                             /* --- UFAPI Input Call Area ---------- */
typedef struct {
   char   pSig[4];                /* Signature 'PDRI'               */

   char   cCallType;              /* Type - Init, Norm, or Term     */

   char   cCallFrom;              /* Called from - R/P rule/pagerule */

   char   pSave1[2];              /*  (future)                      */

   int    iInRC;                  /* Initial RC                     */

   int    iInRV;                  /* Initial RV                     */

   char   pSave2[20];             /*  (future)                      */

   int    iInSize;                /* Size of Input data             */

   char   pInData[2];             /* Input data (blank padded)      */
} UFIN, *PUFIN;

                             /* --- UFAPI Output Call Area --------- */
typedef struct {
   char   pSig[4];                /* Signature 'PDRO'               */

   int    iOutRC;                 /* Return RC                      */

   int    iOutRV;                 /* Return RV                      */

   char   pSave1[24];             /*  (future)                      */

   int    iOutSize;               /* Size of Output data            */

   char   pOutData[2];            /* Output data (blank padded)     */
} UFOUT, *PUFOUT;

                      /* -- User Function subroutine to be called -- */
long int normc(PUFIN pUFIN, PUFOUT pUFOUT) {
  long int i;
                            /* Error exit if signatures not found    */

  if (memcmp(pUFIN->pSig, "PDRI",4)) return(3);
  if (memcmp(pUFOUT->pSig,"PDRO",4)) return(3);

  pUFOUT->iOutRC = pUFOUT->iOutRV = 0;, /* Clear -99s from RC & RV  */
for (pUFOUT->iOutSize=0,i=pUFIN->iInSize-1;
       pUFOUT->iOutSize<pUFIN->iInSize;
       pUFOUT->iOutSize++) {
    pUFOUT->pOutData[pUFOUT->iOutSize] = pUFIN->pInData[i--];
  }
  return (0);                              /* RC=0 -> all ok   */
} /* NORMC */
```

Call Areas in User-Written Functions

Enrichment communicates with the user-written function through data buffers called “call areas”. Programmers do not have to be concerned with these. However, the programmer who writes the user-written function does need to understand the call areas.

When Enrichment executes the function it passes it two memory buffers called the “input call area” and the “output call area”. Enrichment formats these call areas with specific data to ensure validity and appends them with a data area for the input argument (input area) and return string (output area).

> **Note:** The COBOL code in **Example COBOL User-Written Function** on page 90 illustrates these areas and defines their content.

Input Call Area

The Input Call area is a memory block used to pass the *%%IN_VAR* data from Enrichment to the user-written function. (For more information about *%%IN_VAR*, refer to **Calling User-Written Functions** on page 103.) The size of this memory block is set with the *maxin* parameter of the USERFUNCTION declaration. (For more information about USERFUNCTION, refer to **Declaring User-Written Functions** on page 102.)

The layout of the Input Call area is described in the following table:

Table 9: Layout of Input Call Area

Name	Type	Size	Offset	Description
InSig	CHAR	4	0	Input signature, set to "PDRI" by Enrichment
CallType	CHAR	1	4	Call Type. This will be one of the following: • I Initialization Call • N Normal Call • T Termination Call
CallFrom	CHAR	1	5	Section of the rule file from which it is called: • S START • D DOCUMENT • E PRESORTED • P PAGE • F FINISH
future	CHAR	2	6	Reserved
RC	INT	4	8	Current Return Code (from the previous function)
RV	INT	4	12	Current Return Value (from the previous function)
future	CHAR	20	16	Reserved
InSize	INT	4	36	Size of the input data (less than or equal to *maxin*; if less than *maxin*, Enrichment pads *InData* with blanks)
InData	CHAR	n	40	Input data. The user-written function can define multiple fields within this data for ease of processing. The rule file can then prepare one long string with multiple values using concatenation or the RPUT function.

Hints

- Enrichment sets the Input Call Area and the user-written function should not modify it.
- The Input signature should be created by the user function to ensure that the proper block of memory has been passed to it.
- For extended user-written functions, the initialization and termination calls (call types I and T) do not set *InSize* or *InData* and do not use *OutSize* or *OutData*. If the user-written function returns

anything other than a 0 in the Output Call Area RC or RV, Enrichment assumes the initialization/termination failed. If the RC is between 1 and 7, Enrichment issues a warning. Otherwise, Enrichment issues a severe error.

- For extended user-written functions, the init and term calls only occur once: init when the program is loaded (before any rules) and term at the end of processing.

Output Call Area

The Output Call area is a memory block used to pass the result string from the user-written function back to Enrichment. The size of this memory block is set with the *maxout* parameter of the USERFUNCTION declaration. (For more information about USERFUNCTION, refer to **Declaring User-Written Functions** on page 102.)

The layout of the Output Call area is described in the following table.

Table 10: Layout of Output Call Area

Name	Type	Size	Offset	Description
OutSig	CHAR	4	0	Output signature, set to “PDRO” by Enrichment
RC	INT	4	4	Result Return Code (initially set to -99)
RV	INT	4	8	Result Return Value (initially set to -99)
future	CHAR	24	12	Reserved
OutSize	INT	4	36	Size of output data (initially set to 0; must be set less than or equal to *maxout*)
OutData	CHAR	n	40	Output data (initially blanked for *maxout* characters). The user-written function can define multiple fields within this data for ease in processing. The rule file could then retrieve these multiple values using the SUBSTR or RGET functions.

Hints

- Enrichment sets *OutSig* in the Output Call Area to be checked (not changed) by the user-written function.
- Enrichment initializes RC and RV in the Output Call Area to -99. The user-written function should set them to 0 or another valid return code if successful. If RC or RV is still -99 after the user-written function is called, Enrichment assumes that the function did not run and issues a severe error.
- On mainframe, Enrichment checks Register 15 (R15) after the call to the user-written function. R15 should always be 0 if the user-written function ran, even if there was a problem. Use RC and RV

in the Output Call Area for normal return codes. If R15 is non-zero, Enrichment issues a severe error.

- For extended user-written functions, the initialization and termination calls (call types I and T) do not set *InSize* or *InData* and do not use *OutSize* or *OutData*. If the user-written function returns anything other than a 0 in the Output Call Area RC or RV, Enrichment assumes the initialization/termination failed. If the RC is between 1 and 7, Enrichment issues a warning. Otherwise, Enrichment issues a severe error.
- For multiple calls to the same user-written function (that is, functions referenced multiple places within the rule file), the initialization/termination only occurs once. All calls must specify the same language, type, *maxin*, and *maxout* values.

Compiling and Linking User-Written Functions

You must compile and link the user-written functions before you can use them in Enrichment processing.

Compiling and Linking on UNIX

Enrichment for UNIX can only process functions written in C. You must compile and link to create a shared object where the entry point name is the same as the file name.

AIX

For UNIX systems running AIX, you must place the user-written function name in `name.c` and compile and link the function using the following command:

```
cc name.c -o functionName -e functionName -H512 -T512
```

On 64-bit:

```
cc name.c -o functionName -e functionName -H512 -T512 -q64
```

where *functionName* is the name of the function

You must place the name executable in a directory listed in the LIBPATH environment variable. Typically, this will be `.../Enrichment/bin`.

Sun Solaris

For systems running Sun Solaris, you must place the user-written function name in `name.c` and compile and link the function using the following command:

```
cc name.c -o functionName -G -L.../Enrichment/bin
```

where *functionName* is the name of the function

You must place the name executable in a directory listed in the LD_LIBRARY_PATH environment variable. Typically, this will be `.../Enrichment/bin`.

On Sun Solaris you must specify an environment variable for LD_LIBRARY_PATH in order for the user function to be called properly.

HP-UX

For systems running HP-UX, you must place the user-written function name in `name.c` and compile and link the function using the following commands:

```
cc -Ae -c +z name.c
```

On 64-bit:

```
cc -Ae -c +z +DD64 name.c
```

```
ld -b +s -o functionName name.o
```

where *functionName* is the name of the function.

You must place the name executable in a directory listed in the SWVR_LIB environment variable. Typically, this will be `.../Enrichment/bin`.

Compiling and Linking on Linux

Enrichment on Linux can only process functions written in C. To use a user function you must create a shared object where the entry point name is the same as the file name, and place the shared object in a directory listed in the LD_LIBRARY_PATH environment variable. Keep in mind that Linux is case sensitive.

For example, if the function in the C source is spelled "normc", the control file must use "normc", not “NORMC” or “NorMc”.

The following example command creates a shared object with the same name as the function:

```
$ gcc -shared name.c -o functionName
```

If you run Enrichment on a 64-bit Red Hat Enterprise Linux platform, the user function must be compiled and linked differently. This example shows the command line syntax:

```
$ gcc -shared name.c -o functionName -fPID -m32
```

On 64-bit:

```
$ gcc -shared name.c -o functionName -fPID -m64
```

Compiling and Linking on Windows

Enrichment for Windows supports functions written in any language that creates Windows DLLs.

Compiling and Linking on Mainframe

This section contains sample JCL for compiling and linking functions written in COBOL, Assembler, and C. You can also find the sample JCL files in the Enrichment installation.

The following contains the sample JCL necessary to compile and link a COBOL user-written function.

```
//*jobcard
//*********************************************************************
//**UFUNCCOB: Compile and link-edit a COBOL user function
//**
//**Customization instructions:
//** 1. Add a jobcard.
//** 2. Change NORMCOB to the member name of your user function.
//** 3. Change source_pds to the dataset name of the source code.
//** 4. Change loadmod_pds to the dataset name of the load module.
//*********************************************************************
//COB2UCL PROC MEM=''
//*    PROC FOR COBOL COMPILE, LINK
//COB2, EXEC  PGM=IGYCRCTL,PARM='RES,RENT',REGION=1400K
//SYSPRINT DD  SYSOUT=A
//SYSLIN, DD  DSNAME=&&LOADSET,UNIT=SYSDA,DISP=(MOD,PASS),
//             SPACE=(TRK,(3,3)),DCB=(BLKSIZE=80,LRECL=80,RECFM=FB)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSIN    DD  DISP=SHR,DSNAME=source_pds(&MEM)
//LKED, EXEC  PGM=IEWL,PARM='RENT,LIST,XREF,LET,MAP',COND=(5,LT,COB2),
//            REGION=512K
//SYSLIN   DD  DISP=(OLD,DELETE),DSNAME=&&LOADSET
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=loadmod_pds(&MEM),
//             DISP=SHR,DCB=(BLKSIZE=3072)
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPRINT DD  SYSOUT=A
//       PEND  END OF COB2UCL PROC
//*
//STEP1  EXEC COB2UCL,MEM='NORMCOB'
```

The following contains the sample JCL necessary to compile and link an Assembler user-written function.

```
//*jobcard
//**UFUNCASM: Compile and link-edit an assembler user function ****
//**
//**Customization instructions:
//** 1. Add a jobcard.
```

```
//** 2. Change XNAUF to the member name of your user function.
//** Note:  Enrichment distributes two samples: NORMASM and EXTASM.
//**    Source can be found in installhlq.STREAMW.FUNCTION.
//** 3. Change object_pds to the dataset name of an object library.
//** 4. Change source_pds to the dataset name of the source code.
//** 5. Change loadmod_pds to the dataset name of the load library.
//*********************************************************************
//ASM1,  EXEC PGM=ASMA90,PARM='NODECK,OBJECT,TERM',
//  REGION=6M,COND=(5,LE)//SYSPRINT  DD  SYSOUT=*
//SYSPUNCH  DD  SYSOUT=*
//SYSTERM   DD  SYSOUT=*
//SYSIN     DD  DISP=SHR,DSN=source_pds.ASM(XNAUF)
//SYSLIB    DD  DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1    DD  UNIT=VIO,SPACE=(CYL,(10,5)),DSN=&SYSUT1
//SYSLIN    DD  DISP=OLD,DSN=object_pds(XNAUF)
//*
//LINK1  EXEC  PGM=HEWLKED,REGION=512K,COND=(5,LE),
// PARM=('LET,XREF,LIST,MAP,NORENT,REUS,AMODE=24,RMODE=24')
//SYSPRINT  DD  SYSOUT=*
//SYSLIN    DD  DISP=OLD,DSN=object_pds(XNAUF)
//OBJECT    DD  DISP=OLD,DSN=object_pds
//SYSLIB    DD  DISP=SHR,DSN=loadmod_pds
//SYSLMOD   DD  DSN=loadmod_pds(XNAUF),
//         DISP=SHR,DCB=(BLKSIZE=3072)
//SYSUT1    DD  DISP=NEW,UNIT=SYSDA,SPACE=(CYL,(2,2))
//*
```

The following contains the sample JCL necessary to compile and link a C user-written function.

```
//*jobcard
//*********************************************************************
//**UFUNCC: Compile and link-edit a C user function.
//**Customization instructions:
//** 1. Add a jobcard.
//** 2. Change the STEPLIB to include the dataset name of your
//**    C runtime library, if necessary.
//** 3. Change NORMC to the member name of your user function.
//** 4. Change object_pds to the name of an object library dataset.
//** 5. Change source_pds to the name of a source code dataset.
//** 6. Change loadmod_pds to the name of a load library dataset.
//*********************************************************************
//COMPILE EXEC PGM=CCNDRVR,REGION=96M,
//    PARM='OPTFILE(DD:CCOPT)'
//STEPLIB   DD DISP=SHR,DSN=CEE.SCEERUN
//          DD DISP=SHR,DSN=CBC.SCCNCMP
//SYSMSGS   DD DUMMY,DSN=CBC.SCBC3MSG(EDCMSGE),DISP=SHR
//SYSXMSGS  DD DUMMY,DSN=CBC.SCBC3MSG(CBCMSGE),DISP=SHR
//SYSIN     DD DSN=source_pds(NORMC),DISP=SHR
//SYSLIN    DD DSN=object_pds(NORMC),DISP=OLD
//SYSUT10   DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSCPRT   DD SYSOUT=*
```

```
//CCOPT     DD *
  SEARCH(//'CEE.SCEEH.+',//'CBC.SCLBH.+')
  LIST
  SOURCE
  NOLONG
  NOMAR
  NOSEQ
  NOOE
  TARGET(LE)
//SYSUT1    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8    DD    DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9    DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10   DD  SYSOUT=*
//SYSUT14   DD  UNIT=VIO,SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT15   DD  SYSOUT=*
//*
//*-------------------------------------------------------------
//* PRE-LINKEDIT STEP:
//*-------------------------------------------------------------
//PLKED    EXEC PGM=EDCPRLK,PARM='MAP,NOER',COND=(4,LT,COMPILE),
//  REGION=64M
//STEPLIB   DD  DSN=CEE.SCEERUN,DISP=SHR
//SYSMSGS   DD  DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSLIB    DD  DSN=CEE.SCEECPP,DISP=SHR
//SYSIN     DD  DSN=object_pds(NORMC),DISP=SHR
//          DD  DSN=CBC.SCLBSID(IOSTREAM),DISP=SHR
//          DD  DSN=CBC.SCLBSID(COMPLEX),DISP=SHR
//          DD  DSN=CBC.SCLBSID(ASCCOLL),DISP=SHR
//SYSMOD    DD  DSN=&&PLKSET,UNIT=VIO,DISP=(NEW,PASS),
//              SPACE=(32000,(30,30)),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSOUT    DD   SYSOUT=*
//SYSPRINT DD   SYSOUT=*
//SYSIN2    DD  DUMMY
//*
//***************************************************************
//* LINK
//***************************************************************
//LINK   EXEC  PGM=IEWL,COND=((4,LT,COMPILE),(4,LT,PLKED)),
//    REGION=32M,PARM='AMODE=31,MAP,RENT'
//SYSPRINT DD  SYSOUT=*
//OBJECT    DD  DISP=SHR,DSN=object_pds
```

```
//SYSLIB   DD  DISP=SHR,DSN=CEE.SCEELKED
//SYSLMOD  DD  DSN=loadmod_pds(NORMC),
//             DISP=SHR,DCB=(BLKSIZE=3072)
//SYSUT1   DD  DISP=NEW,UNIT=SYSDA,SPACE=(CYL,(2,2))
//SYSLIN   DD  *
 INCLUDE OBJECT(NORMC)
/*
```

Declaring User-Written Functions

To declare a user-written function, place the USERFUNCTION declaration in the rule file (usually in the START section). Once declared, the user-written function is called in the same manner as any other Enrichment rule function.

The syntax for declaring a user-written function is:

```
USERFUNCTION name module [language type maxin maxout buffers]
```

The following table describes the parameters used to declare a user-written function.

Parameter	Description		Default
name	Up to 20 characters that specify a unique name for the function.		None
module	One of the following: • For mainframe systems, up to 8 characters that specify the actual subroutine name link-edited into its own load module of the same name. • For UNIX, any number of characters that specify the actual subroutine name linked as a shared object, where the entry point name is the same as the file name. The file must be in the library path. • For Windows, the value consists of two parts—DLL:function, where "DLL" is the name of the DLL you created and "function" is the name of the function within the DLL. If you specify only one name, it is used for both the DLL and the function.		None
[*language*]	One of the following that specifies the language type used in the function’s subroutine:		C
	C	IBM C/370 or SAS/C for mainframe systems or UNIX C for UNIX systems. Use C for Windows no matter which language you chose to create the DLL.	
	ASM	IBM Assembler (Mainframe only)	

Parameter	Description		Default
	COB[OL]	IBM VS COBOL II (Mainframe only)	
[*type*]	One of the following interface types for the user-written function:		NORMAL
	N[ORMAL]	One call per entry in the rule file, no initialization or termination calls. (The call type in the Input Call area is N for all calls.)	
	E[XTENDE]	Three types of calls: I — Initialization (once after initial program load) N — Normal call (one call per entry in the rule file) T — Termination (once before final remove).	
[*maxin*]	The maximum size of the input data that will be passed to the function. The Input Call Area will be this size plus 40 bytes. 256 is the default *maxin* setting.		None
[*maxout*]	The maximum size of the result returned from the function. The Output Call Area will be this size plus 40 bytes. 256 is the default *maxout* setting.		None
[*buffers*]	One of the following that specifies the type of memory to pass to the function for the Input and Output Call Areas: • A Pass memory above the 16MB line to the function. • B Pass memory below the 16MB line to the function. This parameter is only available on Mainframe.		A

Calling User-Written Functions

The syntax for calling a user-written function in a rule file is:

```
%%Ans = Name(%%IN_VAR)
```

The following table describes the parameters used to call a user-written function.

Parameter	Description
%%Ans	A variable whose value results from the call to the user-written function.
Name	The user-written function name. This must be identical to the name value in the USERFUNCTION command used to declare the function.

Parameter	Description
%%IN_VAR	The variable or constant string to store in InData in the Input Call Area (that is, the data the user-written function will use as input).

In the example rule file shown in the figure below, the user-written function `ims_database` is a COBOL load module `FNIMS2` that uses an account number to look up a client's age in an IMS database. Either `%%Account` or `%%Account2` contains the customer account. `FNIMS2` is a normal user-written function subroutine with an input string (`%%Account` or `%%Account2`) of maximum length 25 bytes that returns an output string (`%%Age`) of up to 3 bytes. The Input Call area is 65 bytes (40 + 25) in length while the Output Call area is 43 bytes (40 + 3) in length.

```
USERFUNCTION ims_database FNIMS2 COBOL N 25 3
IF %%Account <> '' THEN
   %%Age = ims_database(%%Account)
ELSE
   %%Age = ims_database(%%Account2)
ENDIF
```

Hints

- User-written functions always have one argument, *%%IN_VAR*, that can be a variable or a constant string.
- If multiple variables must be passed into the user-written function, you can concatenate them into one long variable.
- If multiple values must be returned by the user-written function, you can extract them from the result variable using the `SUBSTR` or `RGET` built-in functions.

Sample User-Written Functions

Enrichment is shipped with six sample user-written functions which include three normal functions and two extended functions. The three normal functions are `NORMC`, `NORMCOB`, and `NORMASM`. The three extended functions are `EXTC`, `EXTASM` and `RANDNUM`. You can find these functions on the Enrichment installation media.

NORMC	Reverses a sequence of characters and is written in C.
NORMCOB	Reverses an input string and is written in COBOL.
NORMASM	Reverses a sequence of characters and is written in Assembler.
EXTC	Reverses a sequence of characters and is written in C. During the Initialization call, a counter is initialized. In each normal call, the counter is incremented to

	calculate the number of times the function is invoked. In the Termination call, the counter is printed.
EXTASM	Reverses a passed string and is written in Assembler. During the Initialization call, a counter is initialized. In each normal call, the counter is incremented to calculate the number of times the function is invoked. In the Termination call, the counter is printed.
RANDNUM	Generates a random number using the system time function. It is designed as an extended function to ensure any numbers generated are uniquely random. The function takes a high-range number as an input parameter.

Utilities

Enrichment provides several utilities on Mainframe and UNIX to help you develop applications.

Note: These utilities are provided for your convenience only and are not supported by Precisely.

Mainframe Utilities

Enrichment includes several utilities designed to make creating your Enrichment applications easier.

Running Mainframe Utilities

The mainframe utilities consist of load modules and sample JCL. Some utilities also include REXX execs so that you can run the utilities interactively. The REXX execs, load modules, and sample JCL shipped with Enrichment are in the following data sets:

- hlq.STREAMW.EXEC
- hlq.STREAMW.LOAD
- hlq.STREAMW.JCL

where *hlq* is the high-level qualifier in which you installed Enrichment.

Mainframe utilities run in either batch mode or interactive mode, depending upon the utility.

Running Utilities in Batch Mode

Batch mode indicates that the utility is run using JCL. Batch utilities are generally used as a step in your production JCL to prepare a print stream for Enrichment processing. PDRCCA2M, PDRCCM2A, and PDRXCME are generally run in batch mode.

To run a utility in batch mode, customize the job card as well as the STEPLIB and data set names in the sample JCL shipped with the utility and submit it. Each sample JCL contains comments that describe what you need to customize.

Running Utilities in Interactive Mode

Interactive mode indicates that the utility is run from TSO. PDRCMETA, PDRLNADD, and PDRLNSUB are generally run in this mode. Interactive utilities are usually run ad hoc, during the analysis and setup of an Enrichment project.

To run a utility interactively, change the loadmod variable in the REXX exec shipped with the utility so that it points to the data set in which Enrichment is installed, and then run the exec. Each REXX exec contains comments that describe what you need to customize.

Note: The C run-time library must be in your TSO LOADLIB concatenation for you to run any C programs interactively. Consult your system administrator if you have any problems.

In order for TSO to locate the REXX utilities without your specifying the entire data set name, you must allocate the SYSEXEC DD to the data set. So, if your Enrichment high-level qualifier (HLQ) is hlq, the REXX exec data set would be hlq.STREAMW.EXEC. For example, you can invoke the PDRCMETA exec with the fully qualified name:

```
hlq.STREAMW.EXEC(PDRCMETA) metacodefile
```

or you can allocate hlq.STREAMW.EXEC as SYSEXEC and then invoke PDRCMETA directly, as follows:

```
ALLOC FI(SYSEXEC) DA('hlq.STREAMW.EXEC') SHR
PDRCMETA metacodefile
```

In this case, you would not have to reallocate the file for future references during the current TSO/ISPF session. For example:

```
PDRCMETA metacodefile2
```

Note: If you use these utilities often, you may want to put the SYSEXEC allocation into your TSO log-on CLIST. Ask your system administrator if you do not know how to do this yourself.

PDRCCA2M

PDRCCA2M converts ANSI carriage controls to machine carriage controls.

Syntax

PDRCCA2M '*indsn outdsn switches*'

Table 11: Parameters for PDRCCA2M

Parameter	Description	Default
indsn	The input data set name or DD name. Do not use quotation marks. Use the format DD:*DDName* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the input data set name.	REXX: No Default JCL: DD:SYSUT1
outdsn	The output data set name or DD name. Do not use quotation marks. Use the format DD:*DDName* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the output data set name.	REXX: No Default JCL: DD:SYSUT2
switches	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch specifies the record format to which PDRCCA2M will convert the input, as follows: • /F — Fixed. Keep output records as they were in the input print stream. • /V — VBA. Strip blanks from each record in the output print stream.	/F/E
	The second switch specifies how to pad records inserted into the input print stream, as follows: • /A — Use the ASCII hexadecimal character x'20' to pad inserted records. • /E — Use the EBCDIC hexadecimal character x'40' to pad inserted records.	
	The third switch, /T, sets PDRCCA2M to issue trace information for every 1,000 records.	

Example

```
PDRCCA2M 'D966.IN(TEST1) D966.OUT(TEST1) /V/T'
```

In this example, PDRCCA2M is to convert ANSI carriage controls in `D966.IN(TEST1)` to machine carriage controls in `D966.OUT(TEST1)`. PDRCCA2M will convert the record format to VBA (that is, it will remove all spaces from the end of each record) and will issue trace information every 1,000 records. The REXX exec automatically browses the output data set so you can verify the results.

JCL

The following shows the JCL shipped with the PDRCCA2M utility.

```
//*jobcard
//*******************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where    ***
//**               the following are installed at your site.        ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change SYSUT1 and SYSUT2 DDs to be the datasets  ***
//**               for input and output respectively.               ***
//*******************************************************************
//** PDRCCA2M ** Convert ANSI to Machine Channel Controls           ***
//*******************************************************************
//** PDRCCA2M Parameters:                                           ***
//*************  Parm1: Input file name/DD (or * to keep SYSUT1)    ***
//*************  Parm2: Output file name/DD (or * to keep SYSUT2)   ***
//*************  Parm3: Options (no space between):                 ***
//*************    /F = Fixed (keep lines same as input) - DEFAULT ***
//*************    /V = VBA output (strip blanks from each record) ***
//*************    /A = ASCII pad inserted lines (hex 20)           ***
//*************    /E = EBCDIC pad inserted lines (hex 40)-DEFAULT ***
//*************    /T = Trace every 1000 lines                      ***
//*************      Example:  PARM='* * /V/A/T'                    ***
//*************                PARM='DD:IN DD:OUT /T'               ***
//*******************************************************************
//PDRCCA2M EXEC PGM=PDRCCA2M,PARM='* * /V/T'
//STEPLIB  DD DSN=PDR.STREAMW.LOAD,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCBASE,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCLINK,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=inputdsn,DISP=SHR
//SYSUT2   DD DSN=outputdsn,DISP=SHR
//*
```

REXX Code

The following shows the REXX Code shipped with the PDRCCA2M utility.

```
/* REXX ** Convert ANSI CC to machine CC ****************************/
/* Changes: -Created: 02/01/96 DJK                                  */
/* Customize: Change loadmod below to be where Enrichment is       */
/*            installed.                                            */
/********************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRCCA2M)'"
address TSO
```

```
arg in out strip .

parse source . . execname .;  version = '1.0'
if in = '' | out = '' then do
   Call EXPLAIN
   say 'Enter parameters in the order explained.'
   say 'Do not use quotes around dataset names.'
   pull in out strip .   if in = '' | out = '' then
      call ERROR 8, 'Required parameters indsn and outdsn are missing.'
End

/* Allocate input and output files                                    */

   if sysdsn("'"in"'") <> 'OK'  then do
      say "Input file '"in"' not found:" sysdsn("'"in"'")
      exit 4
   end
   "CALL "loadmod" '''"in"'' ''"out"'' "strip"/T'"
   address ISPEXEC "BROWSE DATASET('"out"')"

exit
/*********************************************************************/
/* ERROR - Error exit with message                                   */
/*********************************************************************/
ERROR:
   if arg(2) <> '' then say arg(2)
   if arg(3) <> '' then say arg(3)
   exit arg(1)
/*********************************************************************/
/* EXPLAIN - Self documenting routine                                */
/*********************************************************************/
EXPLAIN:
   say '(c)Precisely'
   say '    'execname' Version' version ' All rights reserved.'
   say
   say 'Function: Convert ANSI CC to machine CC. '
   say
   say 'Syntax:   %'execname' indsn outdsn strip'
   say '          where: indsn is the input dataset (required).'
   say '          Do not include quotes.'
   say '          outdsn is the output dataseot include quotes.'
   say '          strip are any parameters to pass to the '
   say '          load module.'
   return
```

PDRCCM2A

`PDRCCM2A` converts machine carriage controls to ANSI carriage controls.

Syntax

PDRCCM2A '*indsn outdsn switches*'

Parameter	Description	Default
indsn	The input data set name or DD name. Do not use quotation marks. Use the format DD:*DDName* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the input data set name.	REXX: No Default JCL: DD:SYSUT1
outdsn	The output data set name or DD name. Do not use quotation marks. Use the format DD:*DDName* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the output data set name.	REXX: No Default JCL: DD:SYSUT2
strip	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch specifies the record format to which PDRCCM2A will convert the input, as follows: • /F — Fixed. Keep output records as they were in the input print stream. • /V — VBA. Strip blanks from each record in the output print stream.	/F/E
	The second switch specifies how to pad records inserted into the input print stream, as follows: • /A — Use the ASCII hexadecimal character x'20' to pad inserted records. • /E — Use the EBCDIC hexadecimal character x'40' to pad inserted records.	
	The third switch, /T, sets PDRCCM2A to issue trace information for every 1,000 records.	

Example

```
PDRCCM2A 'D966.IN(TEST1) D966.OUT(TEST1) /V/T'
```

In this example, PDRCCM2A converts machine carriage controls in `D966.IN(TEST1)` to ANSI carriage controls in `D966.OUT(TEST1)`. PDRCCM2A will convert the record format to VBA (that is, it will remove all spaces from the end of each record) and will issue trace information every 1,000 records. The REXX exec automatically browses the output data set so you can verify the results.

JCL

The following shows the JCL shipped with the PDRCCM2A utility.

```
//*jobcard
//*********************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where    ***
//**               the following are installed at your site.        ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change SYSUT1 and SYSUT2 DDs to be the datasets  ***
//**               for input and output respectively.               ***
//*********************************************************************
//** PDRCCM2A ** Convert ANSI to Machine Channel Controls            ***
//*********************************************************************
//** PDRCCM2A Parameters:                                           ***
//**************  Parm1: Input file name/DD (or * to keep SYSUT1)   ***
//**************  Parm2: Output file name/DD (or * to keep SYSUT2)  ***
//**************  Parm3: Options (no space between):                ***
//**************    /F = Fixed (keep lines same as input) - DEFAULT ***
//**************    /V = VBA output (strip blanks from each record) ***
//**************    /A = ASCII pad inserted lines (hex 20)          ***
//**************    /E = EBCDIC pad inserted lines (hex 40)-DEFAULT ***
//**************    /T = Trace every 1000 lines                     ***
//**************      Example:  PARM='* * /V/A/T'                   ***
//**************                PARM='DD:IN DD:OUT /T'              ***
//*********************************************************************
//PDRCCM2A EXEC PGM=PDRCCM2A,PARM='* * /V/T'
//STEPLIB  DD DSN=PDR.STREAMW.LOAD,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCBASE,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCLINK,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=inputdsn,DISP=SHR
//SYSUT2   DD DSN=outputdsn,DISP=SHR
//*
```

REXX Code

The following shows the REXX Code shipped with the PDRCCM2A utility.

```
/* REXX ** Convert ANSI CC to machine CC ****************************/
/* Changes: -Created: 02/01/96 DJK                                 */
/* Customize: Change loadmod below to be where Enrichment is      */
/*            installed.                                           */
/*******************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRCCM2A)'"
address TSO
arg in out strip .

parse source . . execname .;  version = '1.0'
if in = '' | out = '' then do
```

```
   Call EXPLAIN
   say 'Enter parameters in the order explained.'
   say 'Do not use quotes around dataset names.'
   pull in out strip .
   if in = '' | out = '' then
      call ERROR 8, 'Required parameters indsn and outdsn are missing.'
End

/* Allocate input and output files                                    */

   if sysdsn("'"in"'") <> 'OK'  then do
      say "Input file '"in"' not found:" sysdsn("'"in"'")
      exit 4
   end
"CALL "loadmod" '''"in"'' ''"out"'' "strip"/T'"

   address ISPEXEC "BROWSE DATASET('"out"')"
exit
/*********************************************************************/
/* ERROR - Error exit with message                                   */
/*********************************************************************/
ERROR:
   if arg(2) <> '' then say arg(2)
   if arg(3) <> '' then say arg(3)
   exit arg(1)

/*********************************************************************/
/* EXPLAIN - Self documenting routine                                */
/*********************************************************************/
EXPLAIN:
   say '(c)Precisely'
   say ',  'execname' Version' version ' All rights reserved.'
   Say
   say 'Function: Convert ANSI CC to machine CC. '
   say
   say 'Syntax:, %'execname' indsn outdsn strip'
   say ', , ,  where: indsn is the input dataset (required).'
   say ', , , , , , , Do not include quotes.'
   say ', , , , ,   outdsn is the output dataset (required).'
   say ', , , , , , , Do not include quotes.'
   say ', , , , ,   strip are any parameters to pass to the '
   say ', , , , , , , load module.'
   Return
```

PDRCMETA

PDRCMETA interprets Metacode in the input print stream to create a readable file.

> **Note:** PDRCMETA produces up to three records of output for each record of input processed. The *orig*, *ascii*, and *expl* parameters control which output records are displayed.

Syntax

PDRCMETA '*indsn outdsn orig ascii expl start numl*'

Parameter	Description	Default
indsn	The input data set name or DD name. Do not use quotation marks. Use the format `DD:`*`DDName`* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the input data set name.	
outdsn	The output data set name or DD name. Do not use quotation marks. Use the format `DD:`*`DDName`* for DDs. In JCL, an asterisk (*) indicates the default DD. In REXX, you must specify the output data set name.	*`hlq`*`.SEEMETA` where *hlq* is your TSO ID or the default qualifier
orig	One of the following: • Y — Display the original data. • N — Do not display the original data.	N
ascii	One of the following: • Y — Display the ASCII data translated into EBCDIC. • N — Do not display the translated ASCII data.	N
expl	One of the following: • Y — Display an explanation of the Metacode data. • N — Do not display an explanation.	Y
start	The number of the first record to display.	1
numl	The number of records to display.	1000000

Example

```
PDRCMETA 'D96.META'
```

In this example, PDRCMETA is to interpret Metacode in the input (`D96.META`) to create the readable output (*`hlq`*`.SEEMETA`). The utility will display only an explanation of the Metacode data.

```
PDRCMETA 'D96.META D96.META.READ N Y Y 1 1000'
```

In the second example, PDRCMETA is to interpret Metacode in the input `D96.META` to create the readable output `D96.META.READ`. The utility will not display the original data but will display ASCII data and an explanation of the Metacode for the first 1,000 records.

The REXX exec automatically browses the output data set so you can verify the results.

JCL

The following shows the JCL shipped with the PDRCMETA utility.

```
//*jobcard
//*******************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where    ***
//**               the following are installed at your site.        ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change SYSUT1 and SYSUT2 DDs to be the datasets  ***
//**               for input and output respectively.               ***
//*******************************************************************
//** PDRCMETA ** View a readable Metacode file                      ***
//** Parameters:                                                    ***
//**************  Parm1: Input file name/DD                         ***
//**************  Parm2: Output file name/DD                        ***
//**************      Example:  PARM='DD:SYSUT1 DD:SYSUT2'          ***
//*******************************************************************
//PDRCMETA EXEC PGM=PDRCMETA,PARM='DD:SYSUT1 DD:SYSUT2'
//STEPLIB  DD DSN=PDR.STREAMW.LOADCRUN,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCBASE,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCLINK,DISP=SHR
//SYSPRINT DD SYSOUT=*//SYSUT1   DD DSN=inputdsn,DISP=SHR
//SYSUT2   DD DSN=outputdsn,DISP=SHR
//*
```

REXX Code

The following shows the REXX Code shipped with the PDRCMETA utility.

```
/* REXX ** Convert ANSI CC to machine CC ****************************/
/* Changes: -Created: 02/01/96 DJK                                 */
/* Customize: Change loadmod below to be where Enrichment is      */
/*            installed.                                           */
/******************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRCCM2A)'"
address TSO
arg in out strip .
```

```
parse source . . execname .;  version = '1.0'
if in = '' | out = '' then do
, Call EXPLAIN
/* REXX ** Create a readable metacode file ************************/
/* Changes: -Created: 02/01/96 DJK                                */
/* Customize: Change loadmod below to be where Enrichment is     */
/*            installed.                                          */
/*****************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRCMETA)'"
address TSO
arg in out l1 l2 l3 start num .
parse source . . execname .;  version = '1.0'
if in = '' | out = '' then do
   Call EXPLAIN
   say 'Enter parameters in the order explained.'
   say 'Do not use quotes around dataset names.'
   pull in out l1 l2 l3 start num .
   if in = '' then
      call ERROR 8, 'Required parameter indsn is missing.'
End
saveout = out
if saveout = '.'       then saveout = 'SEEMETA'
else if saveout <> ''  then saveout = "'"out"'"
else                        saveout = 'SEEMETA'
if out = '.'       then out = 'SEEMETA'
else if out <> ''  then out = "'''"out"'''"
in = "'"in"'"

/* Allocate input and output files                               */
   if sysdsn(in) <> 'OK'  then do
      say 'Input file' in 'not found:' sysdsn(in)
      exit 4
   end
   in = "'"in"'"
   address TSO "CALL "loadmod" '"in out l1 l2 l3 start num"'"
   address ISPEXEC "BROWSE DATASET("saveout")"

exit
/******************************************************************/
/* ERROR - Error exit with message                                */
/******************************************************************/
ERROR:
   if arg(2) <> '' then say arg(2)
   if arg(3) <> '' then say arg(3)
   exit arg(1)
/******************************************************************/
/* EXPLAIN - Self documenting routine                             */
/******************************************************************/
EXPLAIN:
   say '(c)Precisely'
   say '    'execname' Version' version ' All rights reserved.'
   say
   say 'Function: Create a readable metacode file. '
   say
   say 'Syntax:, %'execname' indsn outdsn orig ascii expl start numl'
   say '         where: indsn is the input dataset (required).'
   say '                    Do not include quotes.'
   say '                 outdsn is the output dataset.  Do not include'
   say '                     quotes.  Default is hlq.SEEMETA where'
   say '                     hlq is your TSO id or default qualifier.'
   say '                    Since these are positional arguments if you'

   say '                     specify any other optional arguments and '
   say '                     want to use the default outdsn, you must '
   say '                     use a period (.) for outdsn. '
   say '                  orig is to display the original data. Default:
 N'
   say '                  ascii is to display ascii data. Default: N'
   say '                  expl is to display explanations. Default: Y'
   say '                  start is the starting line. Default: 1'
   say '                numl is the number of records. Default: 1000000'

   say ' '
   say 'The order of the lines in the output is: original data, ascii'
   say 'data, then explanations.  Of course depending on the arguments,
 '
   say 'some of these lines may not appear.'
   Return
```

PDRLNADD

PDRLNADD adds record length bytes to the beginning of each record of a print stream. Use PDRLNADD prior to downloading a print stream from the host for use with Visual Engineer or Enrichment. Metacode and mixed-mode AFP print streams are generally the only print stream types that require record length bytes to identify records.

Note: Refer to Visual Engineer or the <RECORD> tag discussion in the *Enrichment Language Reference Guide* for information on the methods Enrichment can use to identify records.

Syntax

PDRLNADD '*indsn outdsn length format incl startrec num*'

Parameter	Description	Default
indsn	The name of the input print stream.	
outdsn	The name of the output print stream.	

Parameter	Description	Default
length	The length of the length indicator, as follows: • 2 — The first two bytes of each record comprise the length indicator. • 4 — The first four bytes of each record comprise the length indicator.	2
format	The byte order in the length indicator, as follows: • M — Mainframe. The length indicator for each record is coded with the most significant byte first. This is commonly called Big Endian. • P — PC. The length indicator for each record is coded with the least significant byte first. This is commonly called Little Endian.	P
incl	Specifies whether the length indicator includes its own length and the length of the record, as follows: • I — Inclusive. The record length specified in the indicator for each record includes the length value. • E — Exclusive. The record length specified in the indicator for each record does not include the length value.	E
startrec	Specifies the number of the first record to copy from the input to the output.	1
num	Specifies the number of lines to copy from the input to the output.	10

Example

```
PDRLNADD 'D96.INPUT(SAMPLE) D96.OUTPUT(SAMPLE) 2 M I'
```

In this example, PDRLNADD is to add a two-byte inclusive record length indicator to each record in `D96.INPUT(SAMPLE)`. The record length indicators will be with the coded most significant byte first.

JCL

The following shows the JCL shipped with the PDRLNADD utility.

```
//*jobcard
//*******************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where   ***
//**               the following are installed at your site.        ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change SYSUT1 and SYSUT2 DDs to be the datasets  ***
//**               for input and output respectively.               ***
```

```
//**********************************************************************
//** PDRLNADD ** Add record length indicators to metacode          ***
//**********************************************************************
//** PDRLNADD Parameters:                                          ***
//**************  Parm1: Input file name/DD (or * to keep IN)      ***
//**************  Parm2: Output file name/DD (or * to keep OUT)    ***
//**************  Parm3: Options (delimited by a space):           ***
//**************         len can be 2 or 4 to designate the length
//**************             of the length indicator. Default: 2
//**************         typ can be P for PC or M for mainframe.
//**************             Default: P
//**************         inc can be I (include) or E (exclude).
//**************             Default: E
//**************         start is the starting line. Default: 1
//**************        numl is the number of records. Default: 1000000
//**************      Example:  PARM='DD:IN DD:OUT'
//**************                PARM='DD:IN DD:OUT 2 M I 1 2000'
//**********************************************************************
//PDRLNADD EXEC PGM=PDRLNADD,PARM='DD:IN DD:OUT'
//STEPLIB  DD DSN=PDR.STREAMW.LOAD,DISP=SHR
//         DD DSN=SYS3.CLIB22.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//IN       DD DSN=inputdsn,DISP=SHR
//OUT      DD DSN=outputdsn,DISP=SHR
//*
```

REXX Code

The following shows the REXX Code shipped with the PDRLNADD utility.

```
/* REXX ** Add length indicators to Metacode  ***********************/
/* Changes: -Created 03Jan96 DRBallard                              */
/* Customization: Change the loadmod variable to be the dataset     */
/*                where Enrichment is installed.                  */
/*******************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRLNADD)'"
address TSO
arg in out len sb incl start numlines .
parse source . . execname .;  version = '1.0'
if in = '' | out = '' then do
   Call EXPLAIN
   say 'Enter parameters in the order explained.'
   say 'Do not use quotes around dataset names.'
   pull in out len sb incl start numlines .
   if in = '' | out = '' then
      call ERROR 8, 'Required parameters indsn and outdsn are missing.'
End
In = "'"||in||"'"
Out = "'"||out||"'"

/* Allocate input and output files                                  */
```

```
   if sysdsn(in) <> 'OK'  then
      call ERROR 8, 'Input file' in 'not found:' sysdsn(in)
   say 'Allocating data sets.....'
   dummy = outtrap('tso_out.','*')
   'FREE DDNAME(IN)'
   'FREE DDNAME(OUT)'
   dummy = outtrap('OFF')
   'ALLOCATE DDNAME(IN) DSN('in') SHR'
   if rc <> 0 then
      call ERROR 9, 'Unable to allocate Input file' in
if sysdsn(out) = 'OK' then do
      say out 'already exists.  Do you want to replace it? (Y or N)'
      pull ans
      if left(ans,1) <> 'Y' then call ERROR 4, 'User requested exit'
      'ALLOCATE DDNAME(OUT) DSN('out') SHR'
   end
   else do
     if pos('(',out)>0 then
        'ALLOCATE DDNAME(OUT) DSN('out') SHR'
     else
        'ALLOCATE FILE(OUT) DSN('out') NEW SPACE(30,12) TRACKS ',
           'LRECL(155) BLKSIZE(27998) RECFM(V,B,M)'
   end
   if rc <> 0 then
      call ERROR 10, 'Unable to allocate Output file' out
   "CALL "loadmod" '"in"' '"out"' "len sb incl start numlines"'"
   dummy = outtrap('tso_out.','*')
   'FREE DDNAME(IN)'
   'FREE DDNAME(OUT)'
   dummy = outtrap('OFF')
exit
/*********************************************************************/
/*********************************************************************/
/* ERROR - Error exit with message                                   */
/*********************************************************************/
ERROR:   if arg(2) <> '' then say arg(2)
   if arg(3) <> '' then say arg(3)
   exit arg(1)
/*********************************************************************/
/* EXPLAIN - Self documenting routine                                */
/*********************************************************************/
EXPLAIN:
   say '(c)Precisely'
   say '    'execname' Version' version ' All rights reserved.'
   Say
   say 'Function: Add length indicators to Metacode. '
   say
   say 'Syntax:   %'execname' indsn outdsn len type incl start numl'
   say '          where: indsn is the input dataset (required).'
   say '                     Do not include quotes.'
   say '                 outdsn is the output dataset (required).'
   say '                     Do not include quotes.'
   say '                     len can be 2 or 4 to designate the length'
   say '                         of the length indicator. Default: 2'
   say '                     type can be P for PC or M for mainframe.'
   say '                          Default: P'
   say '                      incl can be I (include) or E (exclude).
Default:E'
   say '                      start is the starting line. Default: 1'
   say '                   numl is the number of records. Default: 1000000'

   return
```

PDRLNSUB

PDRLNSUB removes record length bytes from the beginning of each record of a print stream. Use PDRLNSUB if you are uploading a print stream that contains these record length bytes to the host. Generally, Metacode and mixed-mode AFP print streams contain these bytes.

Syntax

PDRLNSUB '*indsn outdsn length format incl*'

Parameter	Description	Default
indsn	The name of the input print stream.	
outdsn	The name of the output print stream.	
length	The length of the length indicator, as follows: • 2 — The first two bytes of each record comprise the length indicator. • 4 — The first four bytes of each record comprise the length indicator.	2
format	The byte order in the length indicator, as follows: • M — Mainframe. The length indicator for each record is coded with the most significant byte first. This is commonly called Big Endian. • P — PC. The length indicator for each record is coded with the least significant byte first. This is commonly called Little Endian.	P
incl	Specifies whether the length indicator includes its own length and the length of the record, as follows: • I — Inclusive. The record length specified in the indicator for each record includes the length value. • E — Exclusive. The record length specified in the indicator for each record does not include the length value.	E

Example

```
PDRLNSUB 'TEST.META TEST.OUT 2 M I'
```

In this example, each record in the input (TEST.META) contains a two-byte length indicator. The length indicators include their own length and are coded with the most significant byte first. PDRLNSUB is to remove length indicators from the input records and store the result in TEST.OUT.

JCL

The following shows the JCL shipped with the PDRLNSUB utility.

```
//*jobcard
//*******************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where    ***
//**               the following are installed at your site.       ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change UT1 and UT2 DDs to be the datasets        ***
//**               for input and output respectively.               ***
//*******************************************************************
//** PDRLNSUB ** Remove prefixes and unblock records                ***
//*******************************************************************
//** PDRLNSUB Parameters:                                           ***
//**************  Parm1: Input file name/DD (or * to keep UT1)      ***
//**************  Parm2: Output file name/DD (or * to keep UT2)     ***
//**************  Parm3: Options:                                   ***
//**************    /H = help information                           ***
//**************      Example:  PARM='DD:IN DD:OUT /E=L/P=2'        ***
//*******************************************************************
//PDRLNSUB EXEC PGM=PDRLNSUB,
//    PARM=('/E=L /P=2 /F=1 /H DD:UT1 DD:UT2')
//STEPLIB  DD DSN=PDRC.R31.STREAMW.LOADCRUN,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCBASE,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCLINK,DISP=SHR
//SYSPRINT DD SYSOUT=*
//UT1    DD DSN=your.input,DISP=SHR
//UT2    DD DSN=your.output,DISP=SHR
//*
```

REXX Code

The following shows the REXX Code shipped with the PDRLNSUB utility.

```
/* REXX ** Remove prefixes and unblock records. ********************/
/* Changes: -Created 24Apr97 DRBallard                             */
/*          -10Jun97 fix quote problem on the CALL loadmod         */
/* Customization: Change the loadmod variable to be the dataset    */
/*                where Enrichment is installed.                 */
```

```
/*********************************************************************/
loadmod = "'PDR.STREAMW.LOAD(PDRLNSUB)'"
address TSO
arg in out switches
parse source . . execname .;  version = '1.1'
if in = '' | out = '' then do
   Call EXPLAIN
   say 'Enter parameters in the order explained.'
   say 'Do not use quotes around dataset names.'
   pull in out switches   if in = '' | out = '' then
      call ERROR 8, 'Required parameters indsn and outdsn are missing.'
end
in = "'"||in||"'"
out = "'"||out||"'"

/* Allocate input and output files                                  */
   if sysdsn(in) <> 'OK'  then
      call ERROR 8, 'Input file' in 'not found:' sysdsn(in)
   say 'Allocating data sets.....'
   dummy = outtrap('tso_out.','*')
   'FREE DDNAME(IN)'
   'FREE DDNAME(OUT)'
   dummy = outtrap('OFF')
   "ALLOCATE DDNAME(IN) DSN("in") SHR"
   if rc <> 0 then
      call ERROR 9, 'Unable to allocate Input file' in
   if sysdsn(out) = 'OK' then do
      say out 'already exists.  Do you want to replace it? (Y or N)'
      pull ans
      if left(ans,1) <> 'Y' then call ERROR 4, 'User requested exit'
      "ALLOCATE DDNAME(OUT) DSN("out") SHR"
   end
else do
     if pos('(',out)>0 then
        'ALLOCATE DDNAME(OUT) DSN('out') SHR'
     else
        'ALLOCATE FILE(OUT) DSN('out') NEW SPACE(30,12) TRACKS ',
           'LRECL(155) BLKSIZE(27998) RECFM(V,B,M)'
   end
   if rc <> 0 then
      call ERROR 10, 'Unable to allocate Output file' out
"CALL "loadmod" '"switches" '"in"' '"out"' '"
/* "CALL "loadmod "'" switches in out "'"*/
   dummy = outtrap('tso_out.','*')
   'FREE DDNAME(IN)'
   'FREE DDNAME(OUT)'
   dummy = outtrap('OFF')
exit
/*********************************************************************/
/* ERROR - Error exit with message                                   */
/*********************************************************************/
ERROR:
   if arg(2) <> '' then say arg(2)
   if arg(3) <> '' then say arg(3)
   exit arg(1)
/*********************************************************************/
/* EXPLAIN - Self documenting routine                                */
/*********************************************************************/
EXPLAIN:
   say '(c)Precisely'
   say '    'execname' Version' version ' All rights reserved.'
   say
   say 'Function: Remove prefixes and unblock records.'
   say
   say 'Syntax:   %'execname' indsn outdsn switches'
   say '          where: indsn is the input dataset (required).'
   say '                      Do not include quotes.'
   say '                 outdsn is the output dataset (required).'
   say '                      Do not include quotes.'
   say '                 switches are 1 or more parameters from:  '
   say '                      /c /e /f /h /i /n /o /p '
   say '                 /c=EBCDIC|ASCII,   /e=Big|Little|Numeric'
   say '                 /f=1|M|S           /h (help) '
   say '                 /i                 /n '
   say '                 /o=1|2             /p=2|4'
   return
```

PDRXCME

PDRXCME converts DJDE print streams that use CMEs into print streams that use a font index. It is easier for Enrichment to add new records with new fonts, such as barcodes, to a print stream that uses a font index.

Syntax

PDRXCME '*indsn outdsn setdsn switches*'

Parameter	Description	Default
indsn	The input data set name. Do not include quotes.	DD:SYSUT1
outdsn	The output data set name. Do not include quotes.	DD:SYSUT2
setdsn	The setup data set name. Do not include quotes.	DD:SYSCME

Parameter	Description	Default
switches	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch indicates the record format to which PDRXCME will convert the input, as follows: • /F — Fixed. Keep output records as they were in the input print stream. • /V — VBA. Strip blanks from each record in the output print stream.	/F/M
	The second switch indicates the type of carriage controls the print stream contains: • /A — The print stream contains ANSI carriage controls. • /M — The print stream contains machine carriage controls.	
	The third switch, /T, sets PDRXCME to issue trace information for every 1,000 records.	

Example

```
PDRXCME 'TEST.DATA TEST.OUTPUT' /V/T
```

In this example, PDRXCME is to convert CMEs in the input (TEST.DATA) to font indexes in the output (TEST.OUTPUT). The utility will convert the record format to VBA (that is, it will remove all spaces from the end of each record) and issue trace information every 1,000 records.

JCL

The following shows the JCL shipped with the PDRXCME utility.

```
//*jobcard
//*******************************************************************
//**Customize:  1. Change STEPLIB to point to the datasets where    ***
//**               the following are installed at your site.        ***
//**                - Enrichment load module                       ***
//**                - C runtime library                             ***
//**            2. Change SYSUT1 and SYSUT2 DDs to be the datasets  ***
//**               for input and output respectively.               ***
//*******************************************************************
//** PDRXCME ** Convert Xerox CME type records to FONTINDEX         ***
//*******************************************************************
//** PDRXCME  Parameters:                                           ***
//**************  Parms are not necessary for default processing    ***
//**************  Parm1: Input file name/DD (or * to keep SYSUT1)   ***
//**************  Parm2: Output file name/DD (or * to keep SYSUT2)  ***
//**************  Parm3: Setup file name/DD (or * to keep SYSCME)   ***
```

```
//**************  Parm4: Options (no space between):               ***
//**************   /F - Fixed (keep lines same as input) - DEFAULT  ***
//**************   /V - VBA output (strip blanks from each record)  ***
//**************   /A - ANSI carriage control in column 1           ***
//**************   /M - Machine carriage control in col 1 - DEFAULT ***
//**************   /T - Trace every 1000 lines                      ***
//**************      Example:  PARM='* * * /V/A/T'                 ***
//**************                PARM='DD:IN DD:OUT DD:CMEIN /T'     ***
//*******************************************************************
//** Updates:                                                       ***
//** Version 1.1 09/24/96 - Add font index to non-print records     ***
//*******************************************************************
//PDRCCM2A EXEC PGM=PDRXCME,PARM='* * * /V/T'

//STEPLIB  DD DSN=PDR.STREAMW.LOAD,DISP=SHR
//         DD DSN=SYS3.CLIB22.SEDCBASE,DISP=SHR
```

UNIX/Linux Utilities

Running UNIX/Linux Utilities

All of the UNIX/Linux utilities run interactively from the command line or can be included in a shell script.

`unblock` and `block` are generally executed before and after a production Enrichment run. These utilities allow Enrichment to process blocked print streams, a capability not built into the program.

`dumpafp` is generally run ad hoc during the analysis and setup of an Enrichment project.

block

`block` adds a blocking indicator to the beginning of record blocks in a print stream. All records in the print stream must have record length indicators. Use `block` when you need record blocking in a print stream (for example, when you have used unblock prior to running Enrichment and need blocking to print). To use block to support the IBM record and block prefix, specify –e=I as a command line parameter. Do not specify –b, -i or -p.

Note: All UNIX/Linux utility names must be lowercase.

Syntax

block [-b= -e= -h -i –p=] *inputfile outputfile*

Parameter	Description	Default
-b=	The maximum blocking size. Records will be accumulated sequentially into blocks, and no block will exceed the maximum blocking size. Processing will terminate if any record exceeds the maximum blocking size.	Each record will be blocked.
-e=	The byte order in both the blocking length and record length indicator, as follows: • B — The length indicator is coded with the most significant byte first. This is commonly called Big Endian. • L — The length indicator is coded with the least significant byte first. This is commonly called Little Endian. • N — Numeric • I — IBM	L
-h	Displays help for the utility.	
-i	The blocking indicator includes its own length and the length of the block of records. If you do not use -i, the blocking indicator includes only the length of the block of records.	
-p=	The length of both the blocking indicator and record length indicator, as follows: • 2 — A two-byte length is used for record length indicators and will also be used for blocking length indicators. • 4 — A four-byte length is used for record length indicators and will also be used for blocking length indicators.	2
inputfile	The name of the input file.	
outputfile	The name of the output file.	

Example

```
block -e=L -p=4 -i p2pef.in p2pef.out
```

In this example, block is to add four-byte blocking indicators to groups of records in the input `p2pef.in`. The blocking indicators will include their own length and are coded least significant byte first.

dumpafp

`dumpafp` interprets the AFP codes in an AFPDS or AFP mixed print stream to create a readable file.

Syntax

```
dumpafp [-a -c -e -h -i -n -p -t –v -x] file
```

Note: The following `dumpafp` parameters are common switches that perform the same functions as parameters of the <RECORD> tag: -a, -c, -e, -i, -p, -t. Refer to the *Enrichment Language Reference Guide* for more information.

Parameter	Description	Default
-a=	One of the following to determine if AFP records are terminated: • YAFP records are terminated with characters specified in the -t parameter. • N — AFP records are not terminated.	Y
-c=	The text format of the print stream, as follows: • E — Text records use the EBCDIC character set. • A — Text records use the ASCII character set.	E
-e=	The byte order in both the blocking length and record length indicator, as follows: • B — The length indicator is coded with the most significant byte first. This is commonly called Big Endian. • L — The length indicator is coded with the least significant byte first. This is commonly called Little Endian.	B
-h	Displays help for the utility.	
-i	Indicates that the record length indicator includes its own length and the length of the record. If you do not use -i, the record length indicator includes only the length of the record.	
-n	Numbers the output lines.	
-p=	The length of the record length indicator, as follows: • 2 — The first two bytes of each record comprise the record length indicator. • 4 — The first four bytes of each record comprise the record length indicator.	2

Parameter	Description	Default
-t=	Specifies the terminator character(s) (hexadecimal codes), separated by commas.	DOS: D,A UNIX: A
-v	Displays the verbose mode for AFP explanations.	
-x	Indicates that transparent text in PTX records is in ASCII format.	
file	The name of the input file.	

Example

```
dumpafp -c=A -t=A unix.afp
```

In this example, `dumpafp` is to display the input `unix.afp`. The arguments indicate that the input file is in ASCII format and that each record is terminated with X'0A' (an ASCII line feed).

etoa

`etoa` converts files from the ASCII character set to the EBCDIC character set, or vice versa.

Syntax

`etoa` *`inputfile outputfile`* `[AE|EA]`

Parameter	Description	Default
inputfile	The name of the input file.	None
outputfile	The name of the output file.	None
[AE|EA]	One of the following: • AE — Input file will be translated from ASCII to EBCDIC. • EA — Input file will be translated from EBCDIC to ASCII.	EA

Example

```
etoa input.ascii input.ebcdic AE
```

This will translate the file (`input.ascii`) to the output file (`input.ebcdic`) in EBCDIC format.

pdrcca2m

`pdrcca2m` converts ANSI carriage controls to machine carriage controls.

Syntax

`pdrcca2m` *`inputfile outputfile switches`*

Parameter	Description	Default
inputfile	The name of the input file.	input.lin
outputfile	The name of the output file.	output.lin
switches	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch specifies the record format to which `pdrcca2m` will convert the input, as follows: • /F — Keep output records as they were in the input print stream. • /V — Strip blanks from each record in the output print stream.	/F/E
	The second switch specifies how to pad records inserted into the input print stream, as follows: • /A — Use the ASCII hexadecimal character x'20' to pad inserted records. • /E — Use the EBCDIC hexadecimal character x'40' to pad inserted records.	
	The third switch, /T, sets `pdrcca2m` to issue trace information for every 1,000 records. • /T — Sets `pdrcca2m` to issue trace information for every 1,000 records.	

Example

```
pdrcca2m test1.in test1.out /V/T
```

In this example, `pdrcca2m` is to convert ANSI carriage controls in `test1.in` to machine carriage controls in `test1.out`. `pdrcca2m` will remove all spaces from the end of each record and will issue trace information every 1,000 records.

pdrccm2a

`pdrccm2a` converts machine carriage controls to ANSI carriage controls.

Syntax

`pdrccm2a` *`inputfile outputfile switches`*

Parameter	Description	Default
inputfile	The name of the input file.	input.lin
outputfile	The name of the output file.	output.lin
switches	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch specifies the record format to which `pdrccm2a` will convert the input, as follows: • /F — Keep output records as they were in the input print stream. • /V — Strip blanks from each record in the output print stream.	/F/E
	The second switch specifies how to pad records inserted into the input print stream, as follows: • /A — Use the ASCII hexadecimal character x'20' to pad inserted records. • /E — Use the EBCDIC hexadecimal character x'40' to pad inserted records.	
	The third switch, /T, sets `pdrccm2a` to issue trace information for every 1,000 records. • /T — sets `pdrccm2a` to issue trace information for every 1,000 records.	

Example

```
pdrccm2a test1.in test1.out /V/T
```

In this example, `pdrccm2a` is to convert machine carriage controls in `test1.in` to ANSI carriage controls in `test1.out`. `pdrccm2a` will remove all spaces from the end of each record and will issue trace information every 1,000 records.

pdrxcme

`pdrxcme` converts DJDE print streams that use CMEs into print streams that use a font index. It is easier for Enrichment to add new records with new fonts, such as barcodes, to a print stream that uses a font index.

Syntax

`pdrxcme` *`inputfile outputfile setupfile switches`*

Parameter	Description	Default
inputfile	The name of the input file. Do not include quotes.	input.djd
outfile	The name of the output file. Do not include quotes.	output.djd
setupfile	The name of the setup file. Do not include quotes.	setupfile
switches	Up to three switches with no intervening spaces. A slash must precede each switch. The first switch specifies the record format to which PDRXCME will convert the input, as follows: • /F — Keep output records as they were in the input print stream. • /V — Strip blanks from each record in the output print stream. The second switch indicates the type of carriage controls the print stream contains: • /A — The print stream contains ANSI carriage controls. • /M — The print stream contains machine carriage controls. The third switch, /T, sets `pdrxcme` to issue trace information for every 1,000 records. • /T — sets `pdrxcme` to issue trace information for every 1,000 records.	/F/M

Example

```
pdrxcme TEST.DATA TEST.OUTPUT /V/T
```

In this example, `pdrxcme` is to convert CMEs in the input (`TEST.DATA`) to font indexes in the output (`TEST.OUTPUT`). The utility will remove all spaces from the end of each record and will issue trace information every 1,000 records.

pswrapper

`pswrapper` is a utility program that converts the Mailer's Choice tag file from line data into PostScript.

Syntax

pswrapper *inputfile outputfile* [*fontname fontsize*] [*psswitch*]

Parameter	Description	Default
inputfile	The Mailer's Choice tag file to be converted.	None
outputfile	The name of the output file.	None
fontname	The font to be used in the output file.	Courier
fontsize	The size (in points) of the font in the output file.	12
psswitch	One of the following: • Y — Will send PCL commands to switch the printer into PS mode and then back to PCL when the file is finished printing. • N — Will not send PCL commands to switch the printer into PS mode.	N

unblock

unblock removes the blocking indicator from the beginning of each group of records in a print stream. For example, it can be used prior to running Enrichment because while Enrichment does not recognize record blocking, it does recognize record length indicators

Syntax

unblock [*-e= -h -i -p=*] *inputfile outputfile*

Parameter	Description	Default
-e=	The byte order in both the blocking length and record length indicator, as follows: • B — The length indicator is coded with the most significant byte first. This is commonly called Big Endian. • L — The length indicator is coded with the least significant byte first. This is commonly called Little Endian.	L
-h	Displays help for the utility.	
-i	The blocking indicator includes its own length and the length of the block of records. If you do not use -i, the blocking indicator includes only the length of the block of records.	
-p=	The length of the blocking indicator, as follows: • 2 — Two bytes are used for the blocking indicator. • 4 — Four bytes are used for the blocking indicator.	2
inputfile	The name of the input file.	
outputfile	The name of the output file.	

Example

```
unblock -e=L -p=4 -i p2pef.tst p2pef.out
```

In this example, each record in the input (`p2pef.tst`) contains a four-byte length indicator. The length indicators include their own length and are coded with the least significant byte first. `unblock` is to remove blocking indicators from the input records and place the result in the output `p2pef.out`.

4 - Commonly-Used Features

In this section

Enhancing Content

Enrichment allows you to enhance content by adding, removing, and changing content in a print stream. Some of the most commonly-used content enhancement features are described below.

Replacing Text

You can use the `<FIELD>` tag to direct Enrichment to take several actions on fields, including replacing text. Use the replace action (R) to update the print stream if the value of the associated field variable changes after it is extracted from the document. The variable value could be changed in a number of ways—such as through CASS address cleansing or by an assignment statement in the rules.

For example, if you want to change a field called *%%Account_Number*:

```
<FIELD> %%Account_Number R
```

The new *%%Account_Number* value will automatically replace the old value when the document is written.

Note that the coding above will affect only the first occurrence of *%%Account_Number* on the document. Setting the action parameter to RA will affect all fields located by that `<FIELD>` tag. To change all occurrences, use the following coding:

```
<FIELD> %%Account_Number RA
```

Note: For the example, any occurrences of the account number not preceded by the reference string will not be replaced.

You can format the value of *%%Account_Number* in the rule file so that it prints out properly. Use the `<REPLACE>` tag within the Field group to specify whether the field location can "grow" or "shrink." For information on the `<REPLACE>` tag, see the *Enrichment Language Reference Guide*. The following shows the coding to allow the account number to grow from 12 to 20 characters in length without overwriting other text.

```
<FIELD> %%Account_Number RA
   <REFERENCE> K
   <LOCATION> K
   <REPLACE> 20 Y
</FIELD>
```

Deleting Information

To delete information from a document, use the `<FIELD>` tag's delete action (D) as follows:

```
<FIELD> %%Account_Number D
```

The coding above will cause the first occurrence of the field to be blanked out. The value of *%%Account_Number* will be extracted.

Moving Information

To move an object, use the Add tag group. You must define the object as a field and then extract and delete it. You can then use rules to change the contents of the field before replacing the field information on the page using an Add group. You can also use this method to change the font size; for example, to reduce the size of an address block to allow a POSTNET™ barcode to fit in a window envelope.

Adding Barcodes

Enrichment can add many types of barcodes to a document. Depending on the type of print stream, there are two methods that Enrichment can use to add a barcode: drawn barcodes and font-based barcodes.

Drawn Barcodes

If you print in AFP, Xerox Metacode, PCL, PDF, or PostScript environments, Enrichment can draw the barcode. Drawn barcodes eliminate the need to acquire or modify font resources. To use drawn barcodes, specify the type of barcode you want to add using the `<ADDTYPE>` tag in the Add tag group. For example, the coding shown in the following figure will add a DataMatrix barcode to the last page of documents in an AFP line data print stream.

```
<add>
 <addtype> PDF417            <! PDF417 barcode. >
 <position> 1680 2400 PELS   <! Position. >
 <addpart> 1234567DFAA343GGA <! Value to encode. >
 <orient> 2                  <! Vertical orientation. >
 <onpage> LAST               <! Last page of each document. >
</add>
```

For more information, see the *Enrichment Language Reference Guide* discussion on the `<ADDTYPE>` tag.

Keep the following in mind for drawn barcodes:

- Enrichment can only draw certain barcode types (see **Supported Drawn Barcode Types** on page 138). If you want to add a barcode type that Enrichment cannot draw, use a font-based barcode.
- Using a font for Interleaved 2of5 barcodes requires you to map the characters (generally in a rule file). It is easier to draw this type of barcode.
- Since drawn barcodes do not require a barcode font, there are no size or orientation restrictions.
- Some applications produce line data documents for printing on AFP printers in LINE mode. You must print such documents in PAGE mode to process PTX records added by Enrichment.

Font-Based Barcodes

Font-based barcodes are valid for any print stream type. As specified in the control file, Enrichment uses a font to add the barcode to the document as text. You can add any type of font-based barcode, as long as a corresponding barcode font is available for your printer.

To create a font-based barcode, you must:

- Set the value of a variable to the barcode value
- Add that variable to a document using a barcode font.

> **Note:** Enrichment uses AFP structured fields or Metacode commands to draw OMR marks and barcode lines. If you are not using an AFP printer in PAGE mode or a Metacode printer, you must use `<ADDTYPE>TEXT` in conjunction with the <POSLINE> tag or the Insertrec tag group to add font-based barcodes.

To create a font-based barcode, specify `<ADDTYPE>TEXT` and use the `<FONTNUM>` tag to specify a font that Enrichment will use to print the barcode. For example, the coding shown in the following figure will add a single OMR mark to the last page of documents in an impact print stream.

```
<add>
 <addtype> TEXT                <! OMR using a font. >
 <posline> 1 40 TRC HORIZONTAL <! Goes on first line. >
 <fontnum> 8                   <! Font #8 is the OMR font. >
 <addpart> 1                   <! 1=end of document. >
 <onpage> LLAST                <! last page of last doc only >
</add>
```

Using the Add Tag Group with DJDE

Within impact and DJDE documents, you can use the Add group tags shown below to include text or a barcode by specifying its location (line or record number for text, record number for barcodes) and the font to use.

```
<ADD>
 <ADDTYPE> TEXT
 <ADDPART>                <! several may be required >
 <POSLINE>                <! specify the position in lines or records>
 <FONTNUM>                <! specify the font number >
 <ONPAGE>
</ADD>
```

Impact and DJDE documents require a barcode font to be available to add a barcode. Since the barcode font is not the same as the normal text font, you cannot just put the barcode information on another record in the print stream. As the following shows, the barcode should be added as an overprint record with the correct font.

```
Incorrect:    Correct:
1 MyText...   BARCODE       1 MyText...
                            +6             BARCODE
```

The Add group `<POSLINE>` and `<FONTNUM>` tags allow you to add these barcodes automatically.

Supported Drawn Barcode Types

Enrichment can draw the following types of barcodes:

Interleaved 2OF5

Interleaved 2of5 barcodes contain numbers. The value must contain an even number of digits. If the value does not contain an even number of digits, Enrichment will add a leading zero. The contents of the barcode in a human-readable form (human-readable interface, or HRI) is sometimes included below the barcode. The characters in the HRI are literal translations of the barcode. The inclusion of HRI is optional

000000112234

Matrix 2of5

Matrix 2of5 barcodes contain numbers. The value must contain an even number of digits. If the value does not contain an even number of digits, Enrichment will add a leading zero. This barcode may also contain a check digit. The contents of the barcode in a human-readable form (human-readable interface, or HRI) is sometimes included below the barcode. The characters in the HRI are literal translations of the barcode. The inclusion of HRI in is optional

0123456789

Code 3OF9

Code 3of9 barcodes contain information readable by the optical scanners on common inserters and barcode sorters. The characters valid in a Code 3of9 barcode are:

- The numbers 0 through 9
- The capital letters A through Z
- Dash (-), space, period (.), dollar sign ($), forward slash (/), plus sign (+), percent symbol (%), and asterisk (*).

The contents of the barcode in human-readable form (an HRI) is sometimes included below the barcode. The characters in the HRI are literal translations of the barcode. The inclusion of HRI in a Code 3of9 barcode is optional

* 0 0 0 0 0 0 1 1 2 2 3 4 *

Code 128

Code 128 barcodes can contain the entire ASCII character set. The contents of the barcode may be shown in human-readable form (an HRI) below the barcode.

4-State barcodes/Intelligent Mail Barcodes

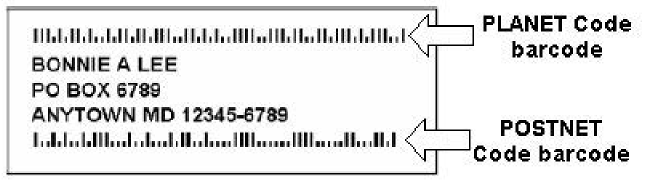
4-State barcodes (called “Intelligent Mail Barcodes” in the U.S.) are used by postal authorities to encode address information so that the mailpiece can be processed using automated equipment that can read the barcode and route the mailpiece accordingly. The 4-State barcode/Intelligent Mail Barcode can also be used for mail tracking.

OMR marks

OMR marks (sometimes called dash marks) are binary on/off marks readable by an optical scanner. They're often used by inserter equipment to control which pages and inserts to put in envelopes. Such marks are the simplest type of barcode and are normally associated with less sophisticated inserters.

PLANET Code® and POSTNET™ Code barcodes

POSTNET™ barcodes are used by USPS® equipment to automate mail processing. PLANET® barcodes are used to uniquely identify each mailpiece in the USPS® mailstream for purposes of tracking the mailpiece. For detailed information on the use of PLANET Code® and POSTNET™ Code barcodes, refer to the USPS® Web site.



DataMatrix

The DataMatrix barcode is a two-dimensional (2D) barcode, meaning that data is encoded both horizontally and vertically. DataMatrix barcodes can encode text, numbers, and data. They can encode more data in a smaller area compared to other barcodes.

PDF417

The PDF417 barcode is a two-dimensional (2D) barcode, meaning that data is encoded both horizontally and vertically. PDF417 barcodes can encode text, numbers and data.

China Post

The China Post barcode is used on mailpieces handled by the Chinese postal authority.

0123456789

MaxiCode

The MaxiCode barcode is a two-dimensional (2D) barcode used primarily on UPS shipping labels. Enrichment supports MaxiCode encoding modes 2 through 6 and has several tags that make it easy to create a MaxiCode barcode for use on a UPS shipping label. These tags start with “UPS” (for example, <UPSPOSTALCODE>).

Aztec Code

The Aztec Code barcode is a two-dimensional (2D) barcode typically used online, on mobile devices, and on digital photographs of documents. The smallest Aztec Code symbol is 15x15 modules square, and the largest is 151x151. The smallest Aztec Code symbol encodes 13 numeric or 12 alphabetic characters, while the largest Aztec Code symbol encodes 3832 numeric or 3067 alphabetic characters or 1914 bytes of data.

QR Code

The QR Code (Quick Response Code) barcode is two-dimensional (2D). The maximum symbol size is 177 modules square, capable of encoding 7366 numeric characters, or 4464 alpha numeric characters. QR Code is designed for rapid reading using CCD array cameras and image processing technology. The information encoded can be made up of four standardized "modes" of data (numeric, alphanumeric, byte/binary, and Kanji), or by supported extensions of virtually any kind of data.

EAN-13

The EAN-13 barcode (originally European Article Number, but now renamed International Article Number) is a 13 digit (12 data and 1 check) barcoding standard which is a superset of the original 12-digit Universal Product Code (UPC) system. The numbers encoded in EAN-13 bar codes are product identification numbers.

Commonly-Used System Variables

The following table lists system variables commonly used to create barcodes.

System Variable	Description
%%CARRIER	The carrier route.
%%DOCUMENT_NO	The overall document number, regardless of which output contains the document.
%%L_PAGE_NO	The page number of the current logical page.
%%L_TOTAL_PAGES	The total number of logical pages in the document.
%%PAGE_NO	The logical front page number of the current page in the document.
%%POSTNET *%%COMP_POSTNET* *%%3800_POSTNET*	Information to place in the POSTNET™ barcode. Typically, *%%POSTNET* is used. However, if you have an IBM 3800 Model 1 printer, use *%%3800_POSTNET* or (if the printer is in Compatibility mode) *%%COMP_POSTNET*. Use *%%POSTNET* if you use an IBM Model 3800 printer in full AFP mode.
%%SECSEG	The sector and segment that comprise the last four characters of a ZIP + 4®.

System Variable	Description
%%SEQUENCE_NO	The sequence number for each document in the file or files identified in an Output tag group.
%%TOTAL_PAGES	The total number of logical front pages in the document.
%%ZIPCODE	Ten characters that indicate the ZIP + 4®.

Adding Barcodes and Other Objects

To add objects such as barcodes, images, text, and TLEs to a print stream, follow these steps. For more information see the *Enrichment Language Reference Guide*.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. If you are adding a barcode or text and the value should be based on data in the print stream, define a field that identifies the data. For example, if you want to create a barcode to represent the account number, define a field to contain the account number. To define a field, use the `<FIELD>` tag group.
3. Decide whether you want to add the object to all outputs or just one output then do one of the following.
   - If you want to add the object to all outputs, create a standalone Add tag group in your control file.
   - If you want to add the object to a specific output, create an Add tag group in the Output tag group where you want to add the object.
4. In the Add tag group, create an `<ADDTYPE>` tag to specify the type of object you want to add (2of5 barcode, PLANET Code® barcode, text, etc).
5. In the Add tag group, create one or more `<ADDPART>` tags to specify the value of the object you are adding. For example, if you are adding a barcode, the tag would specify the numbers that make up the barcode.

   Note: If you are creating a MaxiCode barcode and specify <MAXICODE> with a value of 2 or 3, do not use the <ADDPART> tag to specify the value to encode. Instead, use the tags that begin with “UPS” (for example, <UPSPOSTALCODE>).

6. (AFP, Metacode, PCL, PDF, and PostScript. print streams only)

   In the Add tag group, create a `<POSITION>` tag to specify the horizontal and vertical placement of the object on the page.

7. (Multiple up AFP print streams only)

   In the Add tag group, create a `<POSMULTUP>` tag to specify the horizontal and vertical placement of the object on the page.

8. (Line data only)

   If you are adding a text object (`<ADDTYPE>TEXT`), create a `<POSLINE>` tag in the Add tag group. The tag `<POSLINE>` specifies the horizontal and vertical placement of the text object on the page.

9. (AFPDS print streams only)

   Use the `<SMARTMCFS>` tag to specify whether or not Enrichment should use the font in the MCF1 or MCF2 records when adding text for which no font was specified. This only applies to `<ADDTYPE> TEXT` and barcodes added with HRI text (HRI, or “human-readable interface”, is the text that appears below a barcode).

For information on additional options that you can use when adding barcodes, images, text, and TLEs, refer to the documentation on the Add tag group in the *Enrichment Language Reference Guide*..

Adding Drawn Barcodes to Metacode Documents

You can add drawn barcodes to Xerox Metacode data. Enrichment adds the commands necessary to draw the barcode. Use the Add group tags shown below to create a drawn barcode for Metacode data:

```
<ADD>
 <ADDTYPE>                  <! set to barcode type, example: 3of9>
 <ADDPART>                  <! several may be required to build  >
                            <!      the barcode                  >
 <POSITION>                 <! if not a multiple-up input        >
 <ORIENT>
 <HEIGHT>
 <BARS>
 <ONPAGE>
</ADD>
```

Adding Font-Based Barcodes to Metacode Documents

You can add font-based barcodes to Xerox Metacode data. This is the same as adding text. Enrichment adds the commands necessary to print the lines in the barcode using a font. The appropriate font must be installed on your system or printer. You may need to adjust your Xerox form (FRM) or JSL to include the barcode font. Use the Add group tags shown below to create this type of barcode.

```
<ADD>
 <ADDTYPE> TEXT
 <ADDPART>             <! several may be required            >
 <POSITION>            <! the location to print the barcode   >
 <FONTNUM>             <! specify a barcode font for line data>
 <ONPAGE>
</ADD>
```

Note: For Metacode documents, the font number specified in the <FONTNUM> tag must be defined in the FRM or JSL. If you need to print the HRI as well as the font-based barcode, you must use a second <ADD> tag that defines the HRI font and position. The <ADDPART> tags will be the same as those for the actual barcode.

Adding Barcodes with Insert Records

In DJDE documents, you can include the barcode in an insert record as a second logical page. You can then use the Insertrec tag group to add the logical page to the document. The insert record should include inline variables (for example; *%%barcode*). Enrichment will place the value of the variable into the insert record.

Note: You will need to use the Insertrec tag group to add barcodes if you cannot add overprint records to your print stream.

Examples

The following example demonstrates how to split an input print stream into two print streams based on the number of pages in the document, then adds OMR marks to one output and a DataMatrix barcode to another.

```
<input>
   <name> INPUT                     <! Identifiable name.            >
   <file> DD:INPUT1                 <! Input file name.              >
   <type> AFPL A                    <! AFP line data w/ANSI controls.>
   <doc> T %%AcctNum CHANGE  <! Top of document when Account changes.>
   <field> %%AcctNum KA      <! Find all occurrences.                >
      <ref> ' ' 'Account Number:' 44 <! Reference starts in column 44>
      <loc> 0  2  8              <! Same line as reference, move 2   >
   </field>                      <!   column, get 8 bytes.           >
</input>
<rule>
   <content>                     <! Rule file is instream.           >
    START:
    %%OMRCount = 0
    %%Counter  = 0
    <! If a one page document, set OMR marks and route to that output.>

    DOCUMENT:
    if %%TOTAL_PAGES = 1 then
     %%OMRCount = %%OMRCount + 1          <! setup OMR mark and    >
     if %%OMRCount > 7 then               <! write to first output >
       %%OMRCount = 1
     endif
     <output> Output1
    <! Document must have more than one page. So set the DataMatrix 
barcode>
    <! and route to the appropriate output.                        >
    else
     %%Counter = %%Counter + 1
     %%Doc = JUSTIFY(%%Counter,R,5,0)
     %%Barcode = SUBSTR(%%AcctNum,1,3) | SUBSTR(%%AcctNum,5,4) | %%Doc
     %%ChkDigit = CHECKSUM(%%Barcode)
     <output> Output2
    endif
   </content>
</rule>
<output>
   <name> Output1                <! Identifiable name.             >
   <file> DD:OUTPUT1             <! Output file name.              >
   <add>
      <addtype> OMR              <! OMR mark for 1 page documents   >
      <addpart> 1 1              <! Mark used for Benchmarking      >
      <addpart> %%OMRCount 3    <! Variable set in rule for 3 bytes.>
      <position> .5 6 in        <! Placement of OMR marks.          >
   </add>
</output>
<output>
   <name> Output2                <! Identifiable name.             >
   <file> DD:OUTPUT2             <! Output file name.              >
   <add>
      <addtype> DataMatrix       <! DataMatrix barcode for 2+ page 
documents. >
      <addpart> *                   <! Framing character            >
      <addpart> %%Barcode 12 L ' '  <! 12-bytes, left justified.    >
      <addpart> %%ChkDigit 1 L ' '  <!  1-byte check digit.         >
      <addpart> *                   <! Framing character            >
     <position> .5 10.5 0 0 IN     <! Placement of DataMatrix barcode.>

     <datamatrix> Y 1          <! Square barcode with minimal error 
correction. >
      <cellsize> 12 PELS        <! Width and height of one cell. >
   </add>
</output>
```

The following illustrates how to create a Code 3of9 barcode with framing characters (*) on each end. The example barcode contains the current page number, the total number of pages, the customer number, and the document number.

```
<ADDPART> *                    <! always include framing character>
<ADDPART> %%PAGE_NO 3 R 0      <! a system variable             >
<ADDPART> %%TOTAL_PAGES 3 R 0  <! a system variable>
<ADDPART> %%customer 3 R 0     <! a user-defined field variable>
<ADDPART> %%SEQUENCE_NO 3 R 0  <! a system variable>
<ADDPART> *                    <! always include framing character>
```

You could achieve the same results by building a single variable in a PAGE: rule that contains all of the necessary information, as shown next.

In the control file:

```
<ADDPART> %%barcode 14 R 0
```

In the PAGE: rule:

```
%%barcode = '*'| %%PAGE_NO | %%TOTAL_PAGES | %%customer | %%SEQ_NO | '*'
```

Note: Since page number is part of the barcode, the value of *%%barcode* must be set in a PAGE: rule. Otherwise, you should set the value in a DOCUMENT: rule.

Adding Inserts

An insert is one or more pages of information to add to an existing document. These inserts become part of the document, and Enrichment processes them like any other page. Enrichment assumes that inserts contain printable information. Thus, they are included in document page counts.

A special type of insert, called an “insert record”, contains any number of data records to add to a document. Normally, insert records are used to add printer control commands to create overlays, change fonts, add barcodes, and so on. Enrichment assumes that insert records don’t contain pages of information. Thus, they aren't included in document page counts.

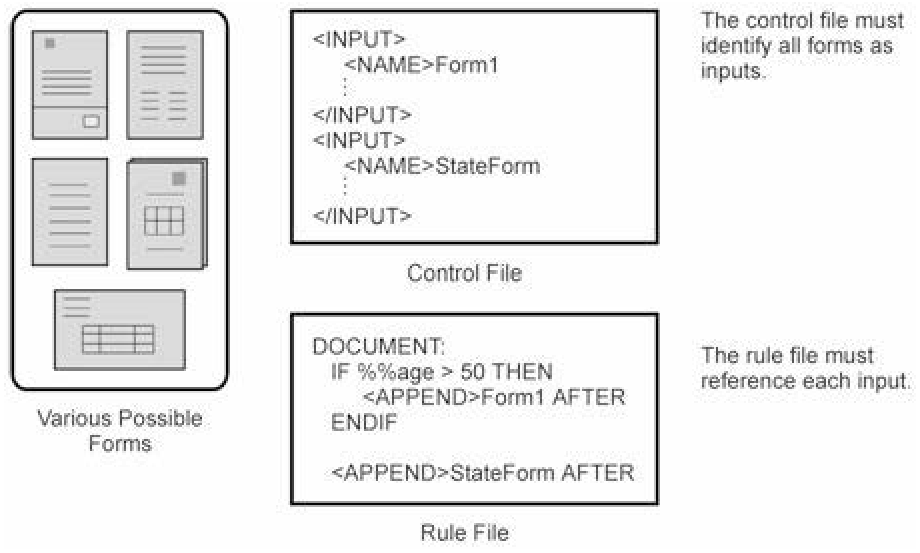
Adding Inserts with Static File Names

Note: Use this method when you only have a limited number of inserts to append, when the inserts have different <TYPE> values, or when the inserts require different AFP or Xerox print resources. You must define each insert in a control file Insertpage group. If you need to select from multiple possible inserts of the same type (and using the same print resources) dynamically, we recommend that you define only one Insertpage group and set the file in the rule file, shown below.

Do the following to append inserts whose file definition will not change:

1. Create the supplemental pages and save them.
2. Use Insertpage tag groups in the control file to identify any of these pages that might be used while processing the application. Within each Insertpage group, specify a `<FILE>` tag that defines the file to be associated with the Insertpage.
3. Use `<APPEND>` commands in the rule file to designate which insert pages to append and where to append them. The order in which Enrichment executes the `<APPEND>` commands in the rule file determines the order in which it assembles the insert pages.

The following diagram illustrates this process.



```
<INPUT>
    <NAME>Form1
    :
</INPUT>
<INPUT>
    <NAME>StateForm
    :
</INPUT>
```

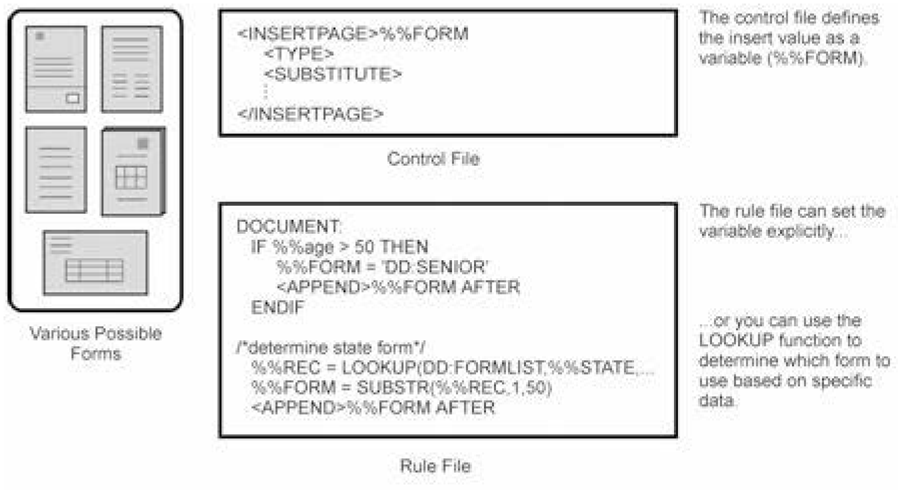
```
DOCUMENT:
  IF %%age > 50 THEN
      <APPEND>Form1 AFTER
  ENDIF

  <APPEND>StateForm AFTER
```

Various Possible Forms
Control File
The control file must identify all forms as inputs.
Rule File
The rule file must reference each input.

Note: If the insert uses COPYGROUP or PAGEFORMAT commands, all files set to the same insert must use the same AFP resources or have fields in the same places.

Adding Inserts with Dynamic File Names

The following diagram illustrates appending an insert. Because the insert value is a variable, Enrichment can specify the file name dynamically, which can greatly reduce the size of your rule file.



```
<INSERTPAGE>%%FORM
    <TYPE>
    <SUBSTITUTE>
    :
</INSERTPAGE>
```

```
DOCUMENT:
  IF %%age > 50 THEN
      %%FORM = 'DD:SENIOR'
      <APPEND>%%FORM AFTER
  ENDIF

/*determine state form*/
  %%REC = LOOKUP(DD:FORMLIST,%%STATE,...
  %%FORM = SUBSTR(%%REC,1,50)
  <APPEND>%%FORM AFTER
```

Various Possible Forms
Control File
The control file defines the insert value as a variable (%%FORM).
Rule File
The rule file can set the variable explicitly...
...or you can use the LOOKUP function to determine which form to use based on specific data.

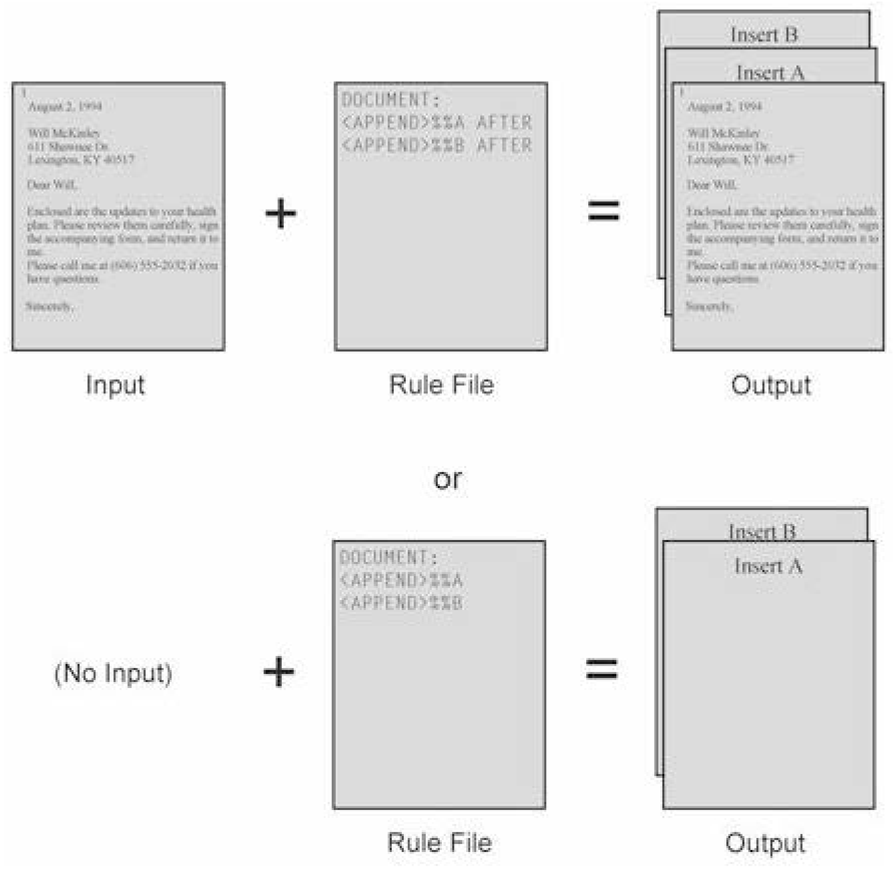
Do the following to append documents in this manner:

Note: If you are adding insert records, replace references to Insertpage and <INSERTPAGE> in this process with Insertrec and <INSERTREC>.

1. Create the supplemental inserts and save them.

2. In the control file, use one set of Insertpage group tags to identify each type of insert you might use while processing the application. The `<INSERTPAGE>` command indicates to Enrichment how to process these documents, regardless of which document is actually being added.
3. Use the `<INSERTPAGE>` tag to specify the name of a variable, such as %%form, whose value is the name of the file to add. The name should be unique for each `<INSERTPAGE>` tag.
4. Set the variable you specified in step 3 to the name of the insert file to add.
5. Use `<APPEND>` commands in the rule file to designate which inserts to add and where to add them. The order in which Enrichment executes the `<APPEND>` commands in the rule file determines the order in which it assembles the documents.

You can use this process to add documents to existing print streams or for document assembly, as shown below.



```
DOCUMENT:
<APPEND>%%A AFTER
<APPEND>%%B AFTER
```

```
DOCUMENT:
<APPEND>%%A
<APPEND>%%B
```

August 2, 1994

Will McKinley
611 Shawnee Dr.
Lexington, KY 40517

Dear Will,

Enclosed are the updates to your health plan. Please review them carefully, sign the accompanying form, and return it to me.
Please call me at (606) 555-2032 if you have questions.

Sincerely,

Specifying the Number of Copies for Inserts

To add more than one copy of an insert to the document, follow the file specification with a comma and the number of copies of the file that you want included. Therefore, on a mainframe system `DD:INPUT, 3` indicates that Enrichment will include three copies of the contents of `DD:INPUT`.

On a UNIX or Windows system, you would specify `/documents/insurance/life, 2` to instruct Enrichment to include two copies of the contents of `/documents/insurance/life`.

For example, assume a bank mails monthly statements to its customers who have a checking account. The rule shown next would include two vouchers for free checks for all customers of retirement age and add an enclosure notice in the regular statement.

```
DOCUMENT:
  <append> %%statement
  IF %%age >= %%retirement_age THEN
     %%enclosure = 'Free check vouchers enclosed.'
     %%IRA = 'DD:INPUT2, 2' // Set the name of the insert with 2 copies

     <append> %%IRA
  ELSE
    %%enclosure = ''
    %%IRA = ''
  ENDIF
```

In this rule, *%%statement*, *%%age*, and *%%retirement_age* are defined by a supplemental data file and all five variables are defined in the control file. Note that *%%age* is compared to *%%retirement_age*, another variable. By keeping the *%%retirement_age* variable current, we ensure that Enrichment always compares the customer age to the current retirement age without changing the rule file.

Using Variable Substitution to Add Inserts

You can use variable substitution with inserts to achieve a variety of results. In this manner you can:

- Force specific pages to be front facing (in duplex printing, for example)
- Add new pages to a document (for marketing, for example)
- Add control information (such as DJDE commands)
- Add barcodes or OMR marks relative to a field
- Build customer document packages

Note: You can only use variable substitution to add inserts with insert templates defined in Insertpage or Insertrec tag groups.

Do the following to use inserts to add supplemental information to your documents:

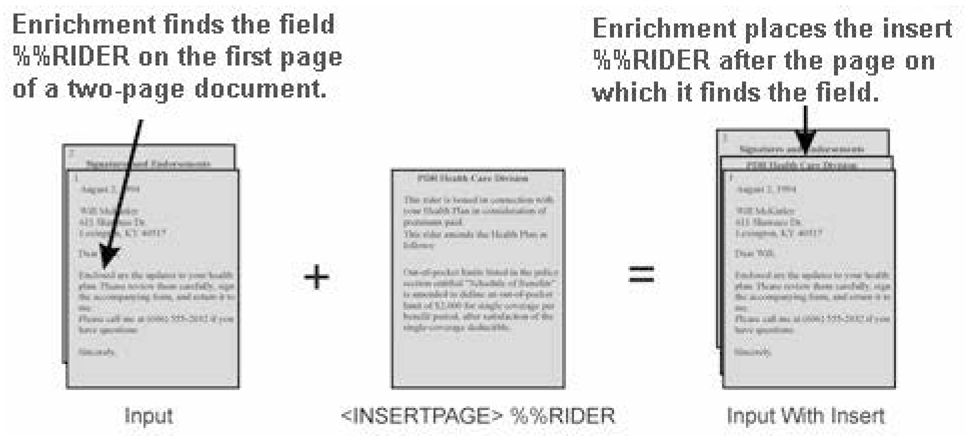
1. Create the supplemental inserts and save them.

   Note: If the insert uses COPYGROUP or PAGEFORMAT commands, all files set to the same insert must use the same AFP resources or have fields in the same places. Similarly, if the insert uses DJDE records, all files set to the same insert must use the same Xerox resources or have fields in the same places.

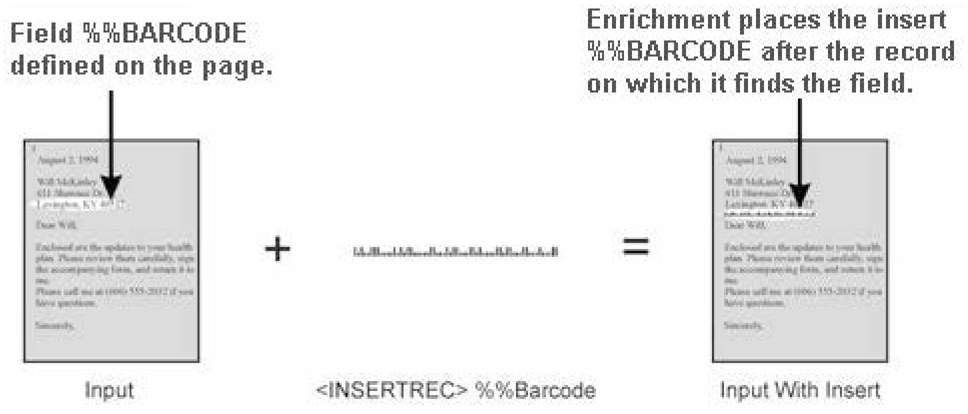
2. In the control file, use one set of Insertpage or Insertrec group tags (whichever is appropriate) to identify each insert. The Insertpage or Insertrec group indicates how Enrichment should process these inserts, regardless of which insert is actually being added. The `<INSERTPAGE>` or `<INSERTREC>` tag maps a file that contains one or more pages or records to a field or variable name. Specify on the appropriate tag the name of a variable, such as *%%form*, whose value is

the name of the file to add. The variable name should be unique for each `<INSERTPAGE>` or `<INSERTREC>` tag.

3. Set the `<PLACE>` tag to *BEFORE* or *AFTER* to place the insert pages or records before or after pages on which the variable is found. Set `<PLACE>` to *WITHIN* to place insert records on the record after the one on which the Enrichment finds the variable.
4. Set the `<SUBSTITUTE>` tag to *YES* in the appropriate Insertpage or Insertrec groups.
5. Use the insert's variable name within the print stream or within supplemental documents to identify where to place the associated file.
6. Set the variable you defined in step 2 to the name of the insert file to add.

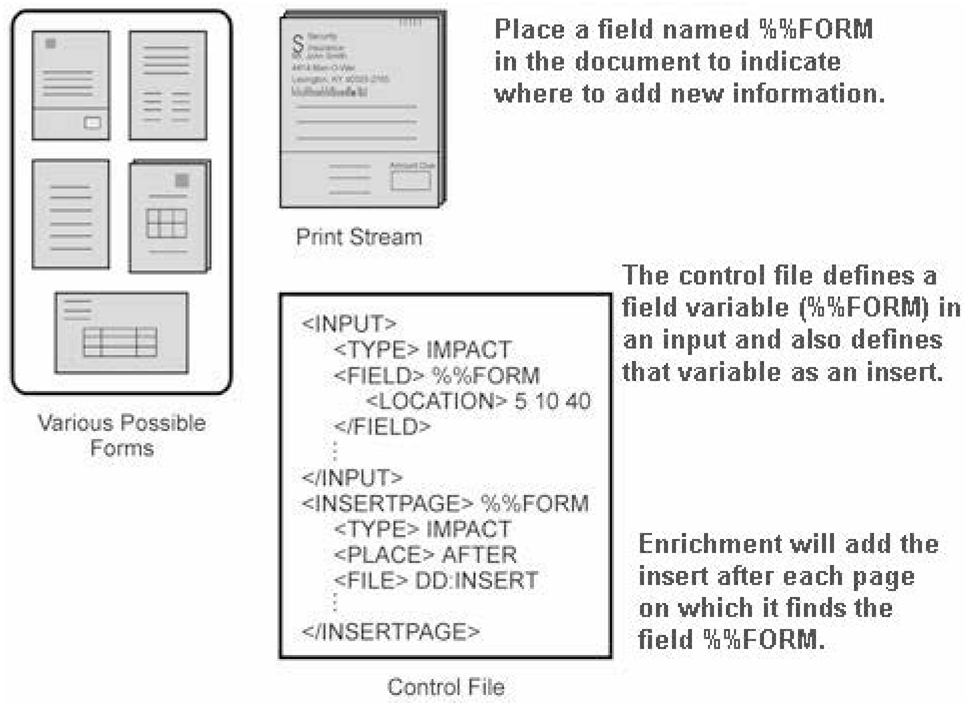

Enrichment finds the field %%RIDER on the first page of a two-page document.
Enrichment places the insert %%RIDER after the page on which it finds the field.
+
=
Input
<INSERTPAGE> %%RIDER
Input With Insert

Enrichment removes the variable names themselves, such as *%%RIDER*, from the document when it adds the insert.


Field %%BARCODE defined on the page.
Enrichment places the insert %%BARCODE after the record on which it finds the field.
+
=
Input
<INSERTREC> %%Barcode
Input With Insert

Using Fields to Specify where to Add Inserts

You can add inserts to a document using field substitution. Field substitution allows you to add the file information at any place in the document at which Enrichment finds a specified field. It behaves exactly like variable substitution, except that the variable needn't be included within the text of the input. The following diagram shows how this works.


Various Possible Forms
Print Stream
Place a field named %%FORM in the document to indicate where to add new information.
The control file defines a field variable (%%FORM) in an input and also defines that variable as an insert.
Enrichment will add the insert after each page on which it finds the field %%FORM.
Control File

```
<INPUT>
   <TYPE> IMPACT
   <FIELD> %%FORM
      <LOCATION> 5 10 40
   </FIELD>
   :
</INPUT>
<INSERTPAGE> %%FORM
   <TYPE> IMPACT
   <PLACE> AFTER
   <FILE> DD:INSERT
   :
</INSERTPAGE>
```

Example

The following example inserts a record (the statement territory) into a field called %%Stmt_Terr, and inserts a page after each document.

```
<input>                            <! Begin input tag group.        >
   <name> Input                    <! File name.                    >
   <file> H:\INPUT1.LIN            <! File location
   <type> I A                      <! Impact with ANSI carriage cntrl>
   <doc> T %%AcctNo CHANGE         <! A change in the account number >
                                   <! denotes the 1st page of a     >
                                   <! document.                     >
   <field> %%AcctNo KA             <! Account number is on line 13, >
      <loc> 13 60 8                <! column 60, for a length of 8. >
   </field>
   <field> %%AmtDue K              <! Amount due field starts on the >
      <loc> 14  60  10             <! 14th print line in position 60 >
   </field>                        <! for 10 characters.            >
   <field> %%Stmt_Terr K           <! Placeholder for insertrec     >
      <loc> 1  1  1 R
   </field>
</input>
<insertpage> %%Award F    <! Customized insert for each document     >
   <file> H:\AWARD.LIN    <! Location of insert                      >
   <type> I A             <! Impact with ANSI carriage cntrl         >
   <substitute> Y         <! variable substitution                   >
</insertpage>
<insertrec> %%Stmt_Terr F  <! Additional insert for each document    >
```

```
   <type> I A               <! Impact with ANSI carriage cntrl        >
   <substitute> Y           <! variable substitution                  >
   <place> WITHIN           <! place after each document              >
   <content>
      Use this territory code with all correspondence: %%Territory
   </content>
</insertrec>
<rule>
   <content>
   START:
      %%Total_Due = 0
      %%Total_Docs = 0
   DOCUMENT:
       %%Territory = SUBSTR(%%AcctNo,1,3)       <! Set %%Territory  >
       %%Total_Docs = %%Total_Docs + 1          <! count invoices & >
       %%Amt = FINDNUM(%%AmtDue,1,2,1,'$',',','.') <! convert field >
                                                <! to numeric.      >
       %%Total_Due = %%Total_Due + %%Amt        <! count amount due.>
       <append> %%Award After
   FINISH:
       <! Format Total into dollar amount and add end banner.      >
       %%Total_Due = FORMAT(%%Total_Due,10,2,Y,0,R,'$',',','.')
       <banner> Totals_Page After
   </content>
</rule>
<output>                              <! Begin output file tag group.>
   <name> Output                      <! File name.                  >
   <file> H:\OUTPUT1.LIN              <! File location.              >
</output>
```

Inserting Records from an External File

To add records from an external file to a print stream, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Create an Insertrec tag group, specifying the variable that triggers the insertion of the record. The record will be inserted on the page, before the page, or after the page where the variable appears, depending on the value you specify in the `<PLACE>` tag (described below). For details of the Insertpage tag group, refer to the *Enrichment Language Reference Guide*.
3. In the Insertrec tag group, create a `<PLACE>` tag to indicate if the insert should be placed within, before, or after the page that contains the variable specified by the `<INSERTREC>` tag. For example, if you want to insert the record after the page that contains the field *%%Closing*, you would specify the following:

```
<INSERTREC> %%Closing
<PLACE>A
```

```
<!Additional Insertrec tag group tags would go here>
</INSERTREC>
```

4. Create a `<FILE>` tag in the Insertrec tag group to specify that file that contains the record you want to insert. For more information, see the *Enrichment Language Reference Guide*.

   Note: If the record that you want to insert is in the control file itself, create a Content tag in the Insertrec tag group. For more information, see the *Enrichment Language Reference Guide*.

5. In the Insertrec tag group, create a `<TYPE>` tag to specify the record's print stream type (AFPDS, Impact, DJDE, etc.).

If the inserted record contains inline variables that you want to replace with values, create a `<SUBSTITUTE>` tag in the Insertrec tag group. For example, if you needed to create the insert shown below and you wanted to replace the text *%%ZIP* with the actual ZIP Code™, you would specify `<SUBSTITUTE>Y`. You would also need to define a field called *%%ZIP* (using the `<FIELD>` tag) that would contain the actual account number extracted from the print stream.

```
ZIP Code: %%ZIP
```

Example

The following example inserts a record that contains the account number.

```
<input>                            <! Begin input tag group.      >
   <name> Input                    <! File name.                  >
   <file> DD:INPUT1                <! DD name in JCL.             >
   <type> I A                      <! Impact with ANSI carriage cntrl>
   <doc> T %%AcctNo CHANGE         <! A change in the account number >
                                   <! denotes the 1st page of a   >
                                   <! document.                   >
   <field> %%AcctNo KA             <! Account number is on line 13,  >
      <loc> 13 60 8                <! column 60, for a length of 8.  >
   </field>
   <field> %%AmtDue K              <! Amount due field starts on the >

      <loc> 14  60  10             <! 14th print line in position 60 >
   </field>                        <! for 10 characters.          >
   <field> %%Stmt_Terr K           <! placeholder for insertrec   >
      <loc> 1  1  1 R   </field>
</input>
<insertrec> %%Stmt_Terr F  <! Additional insert for each document    >
   <type> I A              <! Impact with ANSI carriage cntrl        >
   <substitute> Y          <! variable substitution                  >
   <place> WITHIN          <! place after each document              >
   <content>
      Use this territory code with all correspondence: %%Territory
   </content>
```

```
</insertrec>

<rule>
   <content>
   START:
      %%Total_Due = 0
      %%Total_Docs = 0
   DOCUMENT:
       %%Territory = SUBSTR(%%AcctNo,1,3)     <! Set %%Territory    >

       %%Total_Docs = %%Total_Docs + 1        <! count invoices and >
       %%Amt = FINDNUM(%%AmtDue,1,2,1,'$',',',',','.') <! convert field >
                                                <! to numeric.      >
       %%Total_Due = %%Total_Due + %%Amt        <! count amount due.>
   FINISH:
       <! Format Total into dollar amount and add end banner.       >
       %%Total_Due = FORMAT(%%Total_Due,10,2,Y,0,R,'$',',',',','.')
    </content>
</rule>
<output>                               <! Begin output file tag group.>
   <name> Output                       <! File name.                 >
   <file> DD:OUTPUT1                   <! DD name in JCL.            >
</output>
```

Looking Up Records from a Table or File

You can use data from an external file or a table in the control file for a variety of purposes. For example, you could add an account balance to a statement by looking up the balance in an external file based on the account number on the statement.

To look up records from a table or file, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Determine if the records will be in an external file or if they will be embedded in the control file. If they will be in an external file, continue to the next step. If they will be embedded in the control file, create a Table tag group to contain the data that you want to look up. For more information on the Table tag group, see the *Enrichment Language Reference Guide*.
3. In the control file, create a Rule tag group.
4. Determine which section of the Rule tag group the lookup should happen. This decision is based on when during Enrichment processing the lookup needs to take place. The rule sections are START, DOCUMENT, PRESORTED, PAGE, and FINISH. For more information, see **Developing a Rule File** on page 80.
5. In the appropriate rule section (START, DOCUMENT, PRESORTED, PAGE, or FINISH), create the LOOKUP function to perform the lookup. For more information on the LOOKUP function, see the *Enrichment Language Reference Guide*.

6. Write the appropriate rule section code to use the results from the lookup in the manner you want.

Inserting Banner Pages

A banner page is a page that separates one document from another on a printer. They are sometimes called separator pages. Banner pages are not considered part of any document in the print stream. You can use Enrichment to add banner pages before or after your documents by adding a Banner tag group in the control file to define the pages and using the `<BANNER>` command in your rules to control the placement of the pages.

> **Note:** The banner pages themselves are not created by Enrichment. They must be created by another program in a format consistent with the print streams to which they are added.

To insert banner pages:

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. In the control file, create a Banner tag group to define the banner.
3. In the Banner tag group, create a `<NAME>` tag to define the name to be used when referring to this banner in the control file.
4. In the Banner tag group, create a `<FILE>` tag to specify that file that contains the banner.

   For more information about the <FILE> tag, see the *Enrichment Language Reference Guide*.
5. In the Banner tag group, create a `<TYPE>` tag to specify the banner's print stream type (such as AFPDS, Impact, DJDE).
6. If the banner contains inline variables that you want to replace with values, create a `<SUBSTITUTE>` tag in the Banner tag group.

   For example, if you needed to create the banner shown below and you wanted to replace the text *%%State* with the actual state, you would specify `<SUBSTITUTE>Y`. You would also need to define a field called *%%State* (using the `<FIELD>` tag) that would contain the actual state extracted from the print stream.
7. Because banner pages are not considered part of any other document, they normally are not processed or included in page and document counts. You can control how Enrichment processes banners by setting the Banner group `<ALLOW>` tag.

```
***************************
***************************
*                         *
*                         *
*                         *
*         %%State         *
*                         *
*                         *
*                          *
***************************
***************************
```

8. Create a Rule tag group in the control file.
9. Determine where you want to insert the banner page.

   For example, you may want to insert a banner page whenever the state in the mailing address changes.

10. Create the appropriate Rule logic to insert the banner at the place in the print stream where you want it. To insert the banner use the `<BANNER>` print stream command. For more information see the *Enrichment Language Reference Guide*.

    Note: The <BANNER> print stream command is different from the Banner tag group.

    Note: Always specify the <OUTPUT> command before the <BANNER> command in the rules unless there is only one Output tag group.

   The following is an example Rule that inserts a banner whenever the state changes.

```
<rule>
   <content>
      PRESORTED:
      if CHANGED(%%State) then
        <banner> StateCover BEFORE
      endif
   </content>
</rule>
```

Example

The following example inserts a banner page. The banner page contains a variable for the department which is filled in with the appropriate department name extracted from the document.

```
<input>
   <name> INPUT                     <! Identifiable name.            >
   <file> DD:INPUT                  <! Input file name.              >
   <type> AFPL A       <! AFP Line data with ASCII carriage controls.>
   <doc> T %%AcctNum CHANGE           <! New document when Account   >
                                      <! number changes.             >
   <field> %%AcctNum KA               <! Find account number. Must   >
     <ref> ' ' 'Account Number:' 44   <!  reference because address  >
     <loc> 0 2 8                      <!  is not always on same line >
   </field>
   <field> %%Dept K                   <! Find department on line 3,  >
     <loc> 3 9 3                      <!  column 9, length 3 on first>
   </field>                           <!  page of each document.     >
```

```
</input>
<rule>
   <content>
    PRESORTED:
    <!   Write a banner page each time the department changes. Set  >
    <!   the name of output file to include the sequential number.  >
    if CHANGED(%%Dept) then
       <banner> DEPTCover BEFORE
       %%DSN = "'D966DZB.HANDSON.NEW" | DATE(J) | ".DPT" | %%Dept | "'"

       <filebreak>
    endif
    <output> Output1
   </content>
</rule>
<banner>
   <name> DEPTCover
   <type> I A
   <content>1
       Department %%Dept
   </content>
   <substitute> YES
</banner>
<output>
   <name> Output1                <! Identifiable name.               >
   <dynafile> %%DSN
   <allocate> SYSDA 1 1 TRKS VB 8204 27998
   <filemax> M
   <presort>                     <! Perform system sort.             >
      <pretype> NONE             <! Not LPC or Group1, but other.    >
      <file> DD:INPUTA       <! Indexed sort key file for processing.>
      <sortpart> %%DOCINDEX 8 L ' ' <! Document index system variable>
      <sortpart> %%Dept 3 L ' '     <! Department                    >
      <sortpart> %%TOTAL_PAGES 3 R 0 <! Total pages system variable  >
      <sortpart> %%AcctNum 8 L ' ' <! Account number                 >
   <step> SORT 0 ' SORT FIELDS=(9,3,A,12,3,A,15,8,A),FORMAT=BI,EQUALS
                     OPTION SORTIN=INPUTA,SORTOUT=OUTA'
      <outfile1> DD:OUTA 22         <! Sorted output file.           >
      <indexcol> 1                  <! Document index in column 1.   >
   </presort>
</output>
```

Sorting, Outsorting, and Output

When you sort documents in an input, you change the order of the documents within the input before they are processed by the rules in your rule file. To sort, use the Sortmatch tag group in your control file to identify one or more input print streams to sort. The Sortmatch tag group begins with the

`<SORTMATCH>` tag and ends with the `</SORTMATCH>` tag. Neither tag has parameters, and a control file can contain only one Sortmatch group.

For example, the following example shows coding in which Enrichment will sort documents within the print stream named Statements first by branch office (*%%branch*) and then by customer number (*%%customer*). The resulting output contains documents sorted in descending order by branch office. Within each branch office grouping, Enrichment sorts documents in ascending order by customer number.

```
<SORTMATCH>
<INPUTNAME> Statements
<SORT> %%branch D
<SORT> %%customer A
</SORTMATCH>
```

In this example, the Sortmatch tag group contains these tags:

- `<INPUTNAME>` identifies an input to sort. You can specify as many `<INPUTNAME>` tags as there are Input groups in the control file. For more information, see the *Enrichment Language Reference Guide*.

  Note: Enrichment sorts documents before it processes the rule file, so you can only use variables extracted from the document or associated system variables as sort criteria. If you need to use other data, use the Presort tag group to sort outputs.

- `<SORT>` identifies a field or system variable by which to sort documents within each input and specifies the resulting sort order. You can specify as many `<SORT>` tags as necessary. For more information, see the *Enrichment Language Reference Guide*.

Outsorting carries sorting a step further by separating documents from an input and assigning them to a specific output print stream based on field or system variable values. When outsorting to multiple output files, use an Output tag group to define each output file and then use `<OUTPUT>` commands in a rule file to direct specific documents to specific outputs.

The methods commonly used to sort and outsort output are described below.

Using %%TOTAL_PAGES to Assign Output

The *%%TOTAL_PAGES* system variable's value is the total number of logical front pages in a document. You can use *%%TOTAL_PAGES* in a rule to control which output Enrichment writes a document to based on page count. For example, to send all one-page documents to one output, multi-page documents of 10 pages or less to a second output, and multi-page documents of greater than 10 pages to a third output, you might specify rules as shown below.

```
IF %%TOTAL_PAGES = 1 THEN
   <OUTPUT> ONEPAGE
ELSEIF %%TOTAL_PAGES > 10 THEN
```

```
   <OUTPUT> LARGEOUT
ELSE
   <OUTPUT> MIDOUT
ENDIF
```

Using <OUTPUT> Commands in the Rule File

You can use the rule file to control the output to which Enrichment will write documents. Specify outputs in the rule file as `<OUTPUT>name`, where name references an Output group `<NAME>` tag value in the control file.

You can set `<OUTPUT>NONE` in a rule file to discard documents (that is, to specify that Enrichment does not write the document to any output). To write a document to multiple outputs, set multiple `<OUTPUT>` commands in the rule file.

You can also place multiple copies of a document in a single output by setting multiple `<OUTPUT>` commands to the same name value in the rule.

For example, the rules in the code below indicate that branch representatives will receive copies of statements only for accounts with balances in excess of $20,000. Enrichment will not print statements for accounts with balances of less than $20,000. The rules further separate outputs by using the value of *%%Branch_ID* to identify which of two branches will receive the statements.

```
DOCUMENT:
   IF %%Acct_Balance >= 20000 THEN
   IF %%Branch_ID = AAA THEN
      <OUTPUT> BranchAAA
   ELSEIF %%Branch_ID = BBB THEN
      <OUTPUT> BranchBBB
   ENDIF
   ELSE
      <OUTPUT> NONE
   ENDIF
```

Sorting Documents Within a Single Input

Do the following to sort documents within a single input print stream:

1. Use the Input group `<DOCUMENT>` tag to define the start of each document within the input file.
2. Use a Field tag group within the Input group to define a field for each non-system variable criterion by which you want to sort.
3. Define a Sortmatch tag group.

a. In the `<INPUTNAME>` tag, specify the name of the Input group that defines the print stream to sort. Since the `<INPUTNAME>` tag default is to sort the named file and print it in output, you do not need to specify values for the match and print parameters.
b. Define one `<SORT>` tag for each sort criterion, in the order in which you want the sorts performed. For each `<SORT>` tag, specify the name of a field or system variable by which Enrichment will sort the documents and the order (ASCEND or DESCEND) in which you want the documents placed. If you want the documents placed in ascending order, the Enrichment default, you need only specify the field or system variable name in the `<SORT>` tag.

Sorting an Output Using the Mainframe System Sort

You can use system sort to sort documents going to an output by any field. To invoke system sort, include the Presort tag group with a single `<STEP>` tag that calls the system sort program. For example, when a print stream is to be distributed to both individual agents and individual branches, you can sort the first output by agent and sort the second output by branch. The following shows the Output tag groups that define two outputs, one sorted by agent and one sorted by branch.

Note: This is not a postal presort so no postal presort program, such Mailstream Plus, is necessary. This application uses the mainframe system sort but you could use almost any sort program.

```
<output>                          <! Output sorted by Agent #.  >
  <name> Output1
  <file> DD:OUTPUT1
  <presort>                       <! Presort tag to sort output.>
     <pretype> NONE               <! Use any presort type.      >
     <file> DD:INPUTA
     <sortpart> %%DOCINDEX 9 L ' '  <! Enrichment sort index #  >
     <sortpart> %%AgentNum 7 R 0    <! By Agent # fd using MVS  >
                                    <!  sort utility.           >
      <step> SORT 0 ' SORT FIELDS=(10,7,BI,A),EQUALS
                   OPTION SORTIN=INPUTA,SORTOUT=OUTA'
     <outfile1> DD:OUTA 16          <! Output sorted by Agent # >
     <indexcol> 1 8
  </presort>
</output>
<output>                           <! Output sorted by Branch #.>
  <name> Output2
  <file> DD:OUTPUT2
  <presort>                        <! Presort tag to sort output>
     <pretype> NONE
     <file> DD:INPUTB
     <sortpart> %%DOCINDEX 9 L ' '  <! Enrichment sort index #. >
     <sortpart> %%BranchNum 5 R 0 <! By Branch # fd using MVS   >
                                   <!  sort utility.            >
     <step> SORT 0 ' SORT FIELDS=(10,5,BI,D),EQUALS
                   OPTION SORTIN=INPUTB,SORTOUT=OUTB'
     <outfile1> DD:OUTB 14         <! Output sorted by Branch # >
     <indexcol> 1 8
  </presort>
</output>
```

You can also substitute a user-defined sort program for system sort on the `<STEP>` tag.

Sorting a Print Stream Based on the Order of Another File

The return value for the `LOOKUP` and `LOOKUPV` functions is the number of the record in the internal table that contained the specified string. You can use this value as the basis for sorting a print stream into the order of an external file.

For example, if you create a customer file in the order in which customer statements are to be produced, you could use the code shown below to sort the statements.

```
<INPUT>
   <FIELD>%%CustNo K                <! Pick up customer number.
  >
    .
    .
    .
   </FIELD>
    .
    .
    .
</INPUT>
<RULE>
   <CONTENT>
      DOCUMENT:
         <! LOOKUP the customer number in the customer file
 >
         LOOKUP('DD:CUSTOMER',%%CustNo,1,10)
         IF %%RC THEN                <! Record not found.
 >
            %%ORDERNO = 0
         ELSE
            %%ORDERNO = %%RV
         ENDIF
   </CONTENT>
</RULE>
<OUTPUT>
  .
  .
  .
   <PRESORT>                         <! Sort by %%ORDERNO.
 >
```

```
   .
   .
   .
</OUTPUT>
```

Splitting Output Print Streams Into Multiple Files

An output is associated with a single print stream. This print stream can be written into one or more files. Typically, a single file is used and the name to be created is identified with a single Output group `<FILE>` tag. However, it may be convenient to break an output into multiple files. Often this is done to ensure that each file to be processed has a maximum number of documents or pages. File breaks may also be performed to separate logical units of information (such as file breaks associated with postal tray breaks).

There are two methods for performing convenience breaks:

Automatic breaking based on size

Automatic file breaking is defined by specifying the `<FILEMAX>` tag in the Output group. When you specify `<FILEMAX>` in an output, you will also need to identify each of the files to be created. This can be done with multiple `<FILE>` tags as shown below, or you can use the `<DYNAFILE>` tag.

```
<OUTPUT>
   <FILE> DD:Out1
   <FILE> DD:Out2
   <FILE> DD:Out3
   <FILEMAX> 1000 D
</OUTPUT>
```

The coding shown above results in the placement of documents 1 through 1,000 in `Out1`; 1,001 through 2,000 in `Out2`; and 2,001 through 3,000 in `Out3`. If there are more than 3,000 documents, they are included in `Out3` since there are only three files defined.

Conditional breaking from the rules

You can use the `<FILEBREAK>` command to manually control file breaks in the rules. When Enrichment encounters a `<FILEBREAK>` command in the rules, it does one of the following:

- If you are using dynamic file allocation, Enrichment allocates a new file according to the definition you set up in the Output group `<DYNAFILE>` and `<ALLOCATE>` tags; or
- If you are not using dynamic file allocation, Enrichment moves to the next file specified in multiple `<FILE>` tags.

Note: Unless there is only one output, you must define the output before you use the `<FILEBREAK>` command.

For outputs which are presorted, the `<FILEBREAK>` command must be in the PRESORTED: section of the rules. `<FILEBREAK>` commands in the DOCUMENT: section are ignored.

The `<FILEBREAK>` command causes a file break to occur in the current output. The file break will occur prior to the current document, so that the current document is the first document in the newly allocated file.

You should specify only one file break for each document being processed. If Enrichment encounters multiple `<FILEBREAK>` commands, it ignores all but the last.

The code shown below will cause a file break to occur each time the branch office changes (*%%branchoffice* is a field variable). In this case, we are allocating the new file dynamically.

In the control file:

```
<OUTPUT>
   <NAME> Output
   <DYNAFILE> %%fn
   <ALLOCATE> SYSDA 100
</OUTPUT>
```

In the rules:

```
DOCUMENT:
  <OUTPUT> Output
  IF CHANGED(%%branchoffice) THEN
     %%fn = 'DDZ4.' | %%branchoffice // Define the file name
     <FILEBREAK>           // Break the current output
  ENDIF
```

You can use both automatic and conditional file breaking for the same output. If you use both, the `<FILEMAX>` tag size parameter acts as the default size of the files. You can cause a “premature” break by specifying `<FILEBREAK>`.

For example, the code shown below specifies a maximum size of 1,000 documents, except that the file is also broken when a new branch office occurs.

In the control file:

```
<OUTPUT>
   <NAME> Output
   <FILE> DD:File1
   M
   <FILE> DD:File10
   <FILEMAX> 1000 D
</OUTPUT>
```

In the rules:

```
DOCUMENT:
   <OUTPUT> Output
   IF CHANGED(%%branchoffice) THEN
      <FILEBREAK>                      // Break the current output
   ENDIF
```

Static vs Dynamic File Allocation

When you want to associate multiple files with a single output, you can use one of two methods to specify the files:

- Use multiple `<FILE>` tags to specify the files to be created

  The easiest way to specify multiple files is to use multiple `<FILE>` tags. Enrichment will initially write the print stream to the file associated with the first `<FILE>` tag. When a file break occurs for the output, Enrichment creates the file associated with the next `<FILE>` tag.

- Use dynamic file allocation

  When you use multiple `<FILE>` tags, you must pre-name and pre-allocate each of the output files. This may be inconvenient or unfeasible. Dynamic file allocation allows you to define the name of the file to be created when a file break occurs.

  In dynamic file allocation, the name of the file is assigned by the Output group `<DYNAFILE>` tag and the parameters necessary to set up the sequential data set are specified by the `<ALLOCATE>` tag.

  Note: The `<ALLOCATE>` tag is valid only with Enrichment on mainframe systems. You can't use dynamic file allocation to create files on tape.

Example of Splitting Output Print Streams into Multiple Files

For example, if you wanted to split the documents in the input file into two files, one for documents whose page count is more than three pages (output named `MORE3PAGE`) and one for documents whose page count is three pages or less (output named `3ORLESSPAGE`), you would create the following control file.

```
<input>
   <name> Sortout                      <! Identifiable name.         >
   <file> DD:INPUT1                     <! Input file.                >
   <type> AFPL A                        <! File is AFP line data.     >
   <document> TOP %%Customer_Number CHANGE   <! First page of a doc.  >
   <field> %%Customer_Number KA         <! Find the customer number.  >
      <location> 15 60 10               <! Print line 15, column 60   >
   </field>                             <!   for 10 bytes.            >
</input>
```

```
<rule>
   <content>
   DOCUMENT:
    if %%TOTAL_PAGES > 3 then
       <output> MORE3PAGE
    else
       <output> 3ORLESSPAGE
    endif
   </content>
</rule>

<output>                                      <! Output for 3 or fewer pgs. >
   <name> 3ORLESSPAGE                         <! Identifiable name          >
   <file> DD:OUTPUT1                          <! File name                  >
</output>
<output>                                      <! Output for 3 and up pages. >
   <name> MORE3PAGE                           <! Identifiable name          >
   <file> DD:OUTPUT2                          <! File name                  >
</output>
```

Consolidating Documents

When we sort documents, we place documents in ascending or descending order based on the value of a field. In matching, we consolidate documents that have the same destination so that they can be processed as a package.

To consolidate documents, you use the Sortmatch tag group to sort—and match—documents within multiple inputs. The process for sorting documents within multiple inputs is essentially the same as for a single input.

Note: When multiple inputs are sorted and not matched, they are commingled in the output in sort order. Each input document is treated as a separate document in the output.

Specify a Sortmatch group `<INPUTNAME>` tag for each Input group that defines a print stream to sort. If any of the inputs are already in the desired sort order, you can increase processing efficiency by setting the Input group `<SORTED>` tag to *YES* so Enrichment will not re-sort the input.

After you define all of the necessary `<INPUTNAME>` tags, define one `<SORT>` tag for each sort criterion, in the order in which Enrichment should perform the sorts. Note that you must sort all input streams by the same criteria if you sort them in the same run of Enrichment.

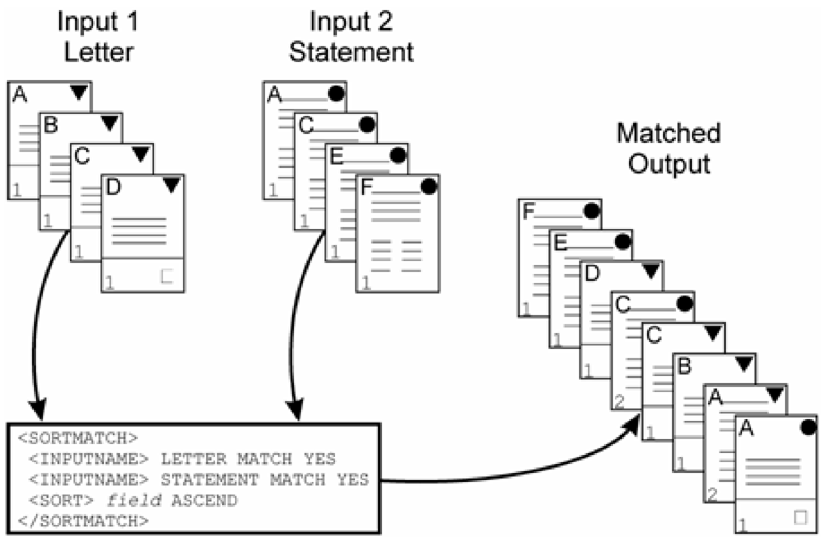
The tagging shown below will cause Enrichment to sort the print streams Statements and Invoices with a primary key of branch office (*%%branch*) and a secondary key of customer number (*%%customer*). The resulting output contains all documents from the inputs Statements and Invoices sorted in descending order by branch office. Within each branch office grouping, documents are sorted in ascending order by customer number.

Note: Since sorting and matching takes place before rule file processing, you cannot set variables in the rule file for use with Enrichment internal sort. You can use these variables in external sorts called during output processing.

```
<SORTMATCH>
   <INPUTNAME> Statements
   <INPUTNAME> Invoices
   <SORT> %%branch D
   <SORT> %%customer A
</SORTMATCH>
```

If you do not want the contents of a particular input to print (for example, if you want to extract data from a matching document without printing that input), set the `<INPUTNAME>` tag match parameter to *YES* and set the print parameter to *NO*.

The diagram below illustrates document consolidation. In the illustration, two input print streams, `LETTER` and `STATEMENT`, are matched so that documents from each input are consolidated and sorted by field. The order of the `<INPUTNAME>` tags indicates the order of pages in the consolidated documents—documents from `LETTER` are page 1 of the consolidated documents and documents from `STATEMENT` are page 2. Documents B and D from `LETTER` and documents E and F from `STATEMENT` aren't consolidated with other documents (because they're not in both inputs), but are sorted into the output.



Using <INPUTNAME> Tag for Consolidation

The `<INPUTNAME>` tag has a required name parameter, which identifies the input print stream whose documents are to be sorted, and two optional parameters, whose functions are as follows:

- The *match* parameter specifies if documents in an input will be sorted or matched then sorted, or if the entire input will be included in every output package. You can set the match parameter to *ALWAYS* (every package includes the entire input), *SORT* (sort the documents in the input by the fields named in `<SORT>` tags), or *MATCH* (sort then match the documents by the fields named in `<SORT>` tags).

- The *printYN* parameter specifies whether to include the input in output. You can set the *printYN* parameter to YES or NO. Normally, you would set print to NO if a document is used only to obtain data and should not be included in a final package.

  Note: You cannot set *printYN* to NO unless you are matching more than one input and you have set the *printYN* parameter for at least one `<INPUTNAME>` tag to YES.

When you consolidate documents from several streams, Enrichment groups the individual documents from each stream together so they can be put into the same envelope.

Note: Consolidation requires that the print streams be sorted by the match keys. If any of the print streams are already in the required sort order, you can improve performance by setting the Input group `<SORTED>` tag to YES for those inputs.

To consolidate multiple streams, use the Sortmatch tag group to identify which streams to consolidate and how to consolidate them. Specify the input streams using Input group tags.

The following code shows how three input streams would be combined and consolidated based on a common customer number. (You must extract the customer number from each document as a field variable and you must name the variable the same—in our example *%%CUSTOMER_NUMBER*—in all three cases).

Note: When you consolidate documents, the combined package is treated as a single document for counting, barcoding, and sorting.

```
<SORTMATCH>
   <INPUTNAME> STREAM1 MATCH YES
   <INPUTNAME> STREAM2 MATCH YES
   <INPUTNAME> STREAM3 MATCH YES
   <SORT> %%customer_NUMBER ASCEND
</SORTMATCH>
```

In the example above, each customer would receive one package containing their documents from each stream in which the customer's number was found.

Enrichment will sort all inputs in ascending customer number order into a single stream. If there are documents from two or more sets that have matching values for *%%CUSTOMER_NUMBER*, Enrichment combines them into a single document in the order in which you specified them in the Sortmatch tag group. Since Enrichment assumes that matched documents go in the same envelope, it counts pages as if the combined entity is a single document. If you need to change barcodes or renumber pages, use Enrichment system variables.

You may need to remove existing barcodes and page numbers from the streams and re-barcode the new combined document. The page numbers and sequence numbers in the original package will be incorrect.

Knowing which Documents were Consolidated

It is often useful to know which print streams were included in a specific output document. There are two methods for determining which print streams are in an output document:

- Using the *%%INPUT* system variable
- Using the FOUND function on unique variables for each print stream

%%INPUT System Variable

The *%%INPUT* system variable contains a blank-delimited list of all inputs used to create the current document. So, in the code below *%%INPUT* would have these values.

Table 12: Example Values for %%Input

Document Set	Inputs Used to Create Document Set
A	"LETTER STATEMENT"
B	"LETTER"
C	"LETTER STATEMENT"
D	"LETTER"
E	"STATEMENT"
F	"STATEMENT"

You can use the WORDPOS function to determine if a specific input is included in the list. For example, the code shown below will add a cover letter to those documents which include a statement.

```
DOCUMENT:
   IF WORDPOS("STATEMENT",%%INPUT) THEN
      <APPEND> CoverLetter B
   ENDIF
```

FOUND Function

The FOUND function identifies whether a specified field was found, or included, in the current document. So, you can use this function to see if unique fields on each input are included in the current output—and thus if the input is included. However, if the field is not found in the document, it doesn't necessarily mean the input isn't present. It could mean the input is present, though the field is not.

Similarly, just checking to see of the field is non-blank can have inconsistent results. The field is reset to blank before each document in which the input is included. If the input isn't included in the current document, the field retains the value from the last document in which the input was included. So it isn't always an accurate indicator of whether an input is present or not.

If you want to check to see what inputs made up the document, use the *%%INPUT* system variable and the WORDPOS function. This is the only truly accurate way to determine what inputs are included in each document.

Consolidating Print Streams

To consolidate print streams, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Create an Input tag group for each input print stream.
3. In the Input tag group, create a <FIELD> tag to define the field that you want to use as the sort key.
4. Create an Output tag group to define each file that you want to commingle the input into. For example, if you wanted to commingle two input print streams into one file you would create one Output tag groups to define the characteristics of the output file. For information on the Output tag group, refer to the *Enrichment Language Reference Guide*.
5. Create a Sortmatch tag group.
6. In the Sortmatch tag group, create an <INPUTNAME> tag for each input print stream that you want to commingle. For example:

```
<SORTMATCH>
<INPUTNAME> MONTHLYSTATEMENTS
<INPUTNAME> YEARENDSTATEMENTS
</SORTMATCH>
```

7. In the Sortmatch tag group, create a <SORT> tag to define the field that you want to use as the sort key. For example, if you want to sort based on the %%AcctNum field, you would enter:

```
<SORTMATCH>
<INPUTNAME> MONTHLYSTATEMENTS
<INPUTNAME> YEARENDTAXSTATEMENT
<SORT> %%AcctNum
</SORTMATCH>
```

Example

The following example includes several different pieces of customer correspondence from a number of inputs in the same envelope to the customer. The example control file reads three print streams that contain multiple documents of varying page count, sorts and matches the documents by customer identification number, and adds a cover letter that includes the customer’s address to each document package. Since every customer will not get every piece of correspondence, we should list the contents of the package in the cover letter. If a document package doesn't include the first input (a provider letter from which we'll extract the customer's address), Enrichment outsorts it to a separate output.

```
<input>                         <! Begin input tag group, 1st input. >
   <name> PL                    <! Name of file.                     >
   <file> DD:INPUT1             <! DD name in JCL.                   >
   <type> AFPL A                <! Linedata w/ ANSI carriage control.>
   <copygroup> F2LDATA0         <! Copygroup for standard output.    >
   <pageformat> P2LINE35        <! Pageformat for standard output.   >
   <doc> T %%ID_number C        <! Change in ID number field denotes >
                                <!    new document.                  >
   <field> %%ID_number KA       <! ID number field, referenced by    >
      <reference> ' ' 'Number:' 17 <!  literal in column 17.         >
      <location> 0 3 10         <!    Actual data is 3 positions to  >
   </field>                     <!    right of the : and is 10 bytes.>
   <field> %%PL_Addr1 K         <! The following 3 fields are the    >
      <loc> 10  2  40           <!    address found on page 1 of the >
   </field>                     <!    document.  They are on lines   >
   <field> %%PL_Addr2 K         <!    10, 11, and 12, beginning in   >
      <loc> 11  2  40           <!    the 2nd position for 40 chars. >
   </field>                     <!    These fields are used in the   >
   <field> %%PL_Addr3 K         <!    cover letter.                  >
      <loc> 12  2  40
   </field>
</input>
<input>                         <! Begin input tag group, 2nd input. >
   <name> PR                    <! Name of file.                     >
   <file> DD:INPUT2             <! DD name in JCL.                   >
   <type> AFPL A                <! Linedata w/ ANSI carriage control.>
   <copygroup> F2LDATA0         <! Copygroup for standard output.    >
   <pageformat> P2LINE35        <! Pageformat for standard output.   >
   <doc> T %%ID_number C        <! Change in id number field denotes >
                                <!    new document.                  >
   <field> %%ID_number KA       <! ID number field on line 9, column >
      <location> 9 26 10        <!   26, for a length of 10.         >
   </field>
</input>
<input>                         <! Begin input tag group, 3rd input. >
   <name> BenS                  <! Name of file.                     >
   <file> DD:INPUT3             <! DD name in JCL.                   >
   <type> AFPL A                <! Linedata w/ ANSI carriage control.>
   <copygroup> F2LDATA0         <! Copygroup for standard output.    >
   <pageformat> P2LINE35        <! Pageformat for standard output.   >
   <doc> T %%ID_number C        <! Change in ID number field denotes >
                                <!    new document.                  >
   <field> %%ID_number KA       <! ID number field on line 5, column >
      <loc> 5 26 10             <!   26, for a length of 10.         >
   </field>
</input>
<input>                         <! Begin input tag group, cover page.>
   <name> CoverLetter           <! Name of file.                     >
   <file> DD:INPUT4             <! DD name in JCL.                   >
   <type> AFPL A                <! Line data w/ANSI carriage control.>
   <copygroup> F2COVLT0         <! Copygroup for cover letter.       >
   <pageformat> P2COVER5        <! Pageformat for cover letter.      >
   <doc> *                      <! Only one document.                >
   <substitute> YES             <! Substitute variable names in the  >
                                <!    cover letter with values from  >
                                <!    the control or rule file.      >
</input>
<sortmatch>                     <! Begin sortmatch tag group.        >
   <inputname> CoverLetter      A Y <! Cover letter always included. >
   <inputname> PL               M Y <! Other files are included only >
   <inputname> PR               M Y <!   if they match.              >
   <inputname> BenS             M Y
   <sort> %%ID_number A         <! Sorted and matched on ID number in>

                                <!   in ascending order.             >
</sortmatch>
<rule>                                 <! Begin rule file.           >
   <content>                           <! Content tag allows rules   >
   DOCUMENT:                           <!    in the control file.    >
     if WORDPOS('PL',%%INPUT) then     <! This block of code checks  >

        %%PL = 'Provider Letter'       <!    for the existence of the>
        %%ADDR1 = %%PL_Addr1           <!    Provider Letter to list >
        %%ADDR2 = %%PL_Addr2           <!    in the cover letter.    >
        %%ADDR3 = %%PL_Addr3
        <output> Output                <! Set the output file.       >
     else
        %%PL = ' '
        %%ADDR1 = '********************'
        %%ADDR2 = '* ADDRESS UNKNOWN *'
        %%ADDR3 = '********************'
        <output> NoAddress             <! Set the output file.       >
     endif
     if WORDPOS('PR',%%INPUT) then     <! This block of code checks  >
        %%PBR = 'Plan Book Revisions'  <!    for the existence of the>
     else                              <!    Plan Book input and adds>
        %%PBR = ' '                    <!    it to the cover letter. >
     endif
     if WORDPOS('BenS',%%INPUT) then    <! This block of code checks >
        %%BS = 'Benefits Summary'      <!    for the existence of the>
     else                              <!    Benefit input and adds  >
        %%BS = ' '                     <!    it to the cover letter. >
     endif
   </content>
</rule>
<output>                          <! Begin output tag group.         >
   <name> Output                  <! Name of file.                   >
   <file> DD:OUTPUT1              <! DD name in JCL.                 >
</output>
<output>                          <! Begin output tag group.         >
   <name> NoAddress               <! Name of file.                   >
```

```
    <file> DD:OUTPUT2             <! DD name in JCL.                  >
</output>
```

Reordering Pages within each Document

To reorder pages within each document, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Create an Input tag group for each input print stream.
3. In the Input tag group, create a <REORDER> tag. This allows you to reverse the order of pages, move the first page of the document so it becomes the last, move the last page of the document so that it becomes the first, or reorder pages for 2-up duplex printing.

Naming Output Files Dynamically

You can dynamically name your output files by following these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Create an Output tag group for each output that you want to create.
3. In the Output tag group, create a <NAME> tag to designate the name that you want to use when referring to this output in the control file.
4. In the Output tag group, create a <DYNAFILE> tag. This tag allows you to use a variable name for the output. For complete information on using the <DYNAFILE> tag, refer to the *Enrichment Language Reference Guide*.

Example

The example control file (shown below) reads a print stream and uses information from a lookup file to add the appropriate department number to each document. Account numbers not found in the lookup file are written to a report file. Documents are then written to separate output files based on the department number.

```
<input>
  <name> INPUT                      <! Identifiable name.            >
  <file> H:\INPUT.AFP               <! Input file name.              >
  <type> AFPL A       <! AFP Line data with ASCII carriage controls.>
  <doc> T %%AcctNum CHANGE            <! New document when Account   >
                                      <! number changes.             >
  <field> %%AcctNum KA
    <loc> 13 60 8                    <! Get 8-byte account number.   >
  </field>
  <field> %%Dept     R               <! Replace the first occurrence >
```

```
    <loc> 5 67 3                       <! on line 5, column 67 for 3   >
  </field>                             <! bytes.                       >
<input>
<rule>
  <content>
   DOCUMENT:
    <!  Build key for file lookup.                        >
    <!  The account number on the file does not include   >
    <!  the dash as it does on the document.              >
    %%Key = SUBSTR(%%AcctNum,1,3) | SUBSTR(%%AcctNum,5,4)
    <!  File lookup for department number.                >
    <!  If the account number is not in the file          >
    <!  a record is written to a sequential file.         >
    %%Record = LOOKUP('H:\LUFILE.TXT',%%Key,1,7,Y)
    if %%RC = 0 then
      %%Dept = JUSTIFY(SUBSTR(%%Record,8,3),L,3,' ')
    else
      %%Dept = '   '
      %%DocNo = JUSTIFY(%%DOCUMENT_NO,R,5,0)
      %%ErrorRecord = %%AcctNum | %%DocNo
      %%err = WRITE('H:\LUERROR.TXT',%%ErrorRecord,VB,8204)
    endif PRESORTED:
     <! Set the name of output file to include the department number. >

    if CHANGED(%%Dept) then
      %%FN = "DEPT" | %%Dept | ".AFP"
      <filebreak>
    endif
    <output> Output1
   </content>
</rule>
<output>
   <name> Output1                    <! Identifiable name.              >
   <dynafile> %%FN
   <filemax> M
   <presort>
      <pretype> NONE                 <! Not LPC or Group1, but other.   >
      <file> H:\INPUTA       <! Indexed sort key file for processing.>
      <sortpart> %%DOCINDEX 9 L ' ' <! Document index system variable>
      <sortpart> %%Dept     3 L ' ' <! Department number              >
      <sortpart> %%AcctNum  8 L ' ' <! Account number                 >
      <step> SORT 0 ' SORT FIELDS=(10,3,A,13,8,A),FORMAT=BI,EQUALS
                      OPTION SORTIN=INPUTA,SORTOUT=OUTA'
      <outfile1> H:\OUTA.AFP 20     <! Sorted output file.            >
      <indexcol> 1                  <! Document index in column 1.    >
   </presort>
</output>
```

Adding a Document to All Documents

If you want to add a particular document to all your other documents, you can use the Sortmatch tag group in the control file to do so. The place at which the document is added is controlled by the order that documents are specified in the Sortmatch group. No rule file is required.

Do the following to add a constant document to all documents:

1. Create the supplemental documents and save them.
2. Use Input tag groups to define the documents.
3. Use the Sortmatch tag group to specify the documents to add.

Example

For example, to add a cover letter to a customer package (built by consolidating STREAM1, STREAM2, and STREAM3), include the cover letter within the Sortmatch tag group. Set the cover letter up as an input. The Sortmatch tag group in the control file would be as shown below.

```
<SORTMATCH>
   <INPUTNAME> COVERLETTER ALWAYS YES
   <INPUTNAME> STREAM1 MATCH YES
   <INPUTNAME> STREAM2 MATCH YES
   <INPUTNAME> STREAM3 MATCH YES
   <SORT> %%customer_NUMBER ASCEND
</SORTMATCH>
```

In the example below, each customer would get a cover letter, their statement, and a marketing piece.

```
<SORTMATCH>
   <INPUTNAME> coverletter ALWAYS          <! Always included
>
   <INPUTNAME> statement MATCH            <! Each individual statement>

   <INPUTNAME> marketing ALWAYS            <! Always included
>
</SORTMATCH>
```

Duplex Output

To create duplex output, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. If the input is already duplex, specify the <DUPLEX> tag in the Input tag group.

3. Create an Output tag group for each output file that you want to create.
4. Specify the <DUPLEX> tag in the Output tag group.

For more information see the *Enrichment Language Reference Guide*.

Example

```
// Sample control file to output a 2-UP  printstream
// input is in line and consists of 2 pages per document
// MUP is done on output and pages are placed horizontally
<input>
    <content>
1  Document 1  Page 1
       more of page 1
1  Document 1  Page 2
       more of page 2
1  Document 2  Page 1
       more of page 1
1  Document 2  Page 2
       more of page 2
    </content>
    <name> INPUT
    <type>I A
    <doc>  T  %%TOPOFDOC C
    <field> %%TOPOFDOC
       <REF> '1'  'Document '
        <loc> 0 1 2
    </field>
    <pagesize> 8.5 11 IN
  <mup>
    <mgrid> 1 1  R C
    <msize> 40 66 8.5 11 IN
  </mup>
</input>

<output>
   <name> OUTPUT
   <file> F:\OUTPUT.LIN
  <mup>
     <mgrid> 2 1  R C
     <msize> 40 66 8.5 11 IN
  </mup>
</output>
```

Multiple-Up Printing

Enrichment converts multiple-up input documents to single-up documents internally in order to count pages, reorder pages, or reorder documents. This conversion occurs before fields, pages, or documents are found. Therefore, you should define all fields with the assumption that all logical pages have been moved to the upper left logical page. To see this translation, simply specify a multiple-up input and a single-up output in your application.

Enrichment can also process multiple-up formats as large single-up pages without converting to individual single-up pages. This technique is efficient, but can be used only if no page counting or reordering of the logical pages is required (for example, in a 2-up page format where each document is one logical page). To do this, process the input as single-up and pick up fields from both the right and left sides of the page. A rule file can process both sides at once.

To create multiple-up output, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. In the Input and/or Output tag group create an Mup tag group. The Mup tag group identifies parameters for processing multiple-up inputs and outputs. Each Input and Output tag group can contain one Mup group. For more information see the *Enrichment Language Reference Guide*.

   Note: The Mup tag group is not valid for Metacode print streams.

FORMDEFs and N-Up Processing

AFP printing allows the print software to perform multiple-up processing by setting up two or three partitions (or logical pages) on the real physical page. These partitions act as physical pages so that a page-eject control (in AFP line data or AFPDS) simply moves printing to the next partition. IBM refers to this process as N-Up processing. N-Up processing requires changes to the FORMDEF resource to establish the partitions. You can only specify two or three partitions on each side of a physical page and each partition must be of equal size. Enrichment processes this type of data as single-up because the data is single-up—the print resources perform the multiple-up processing.

An extension to N-Up processing extends the partition concept so that partitions can vary in different size and there can be a different number of partitions on the front and back of a duplex page. This capability is limited to particular printers that have an enhanced hardware controller. IBM refers to this type of processing as Enhanced N-Up or power-positioning. Enrichment processes Enhanced N-Up as single-up since the data is single-up. You can use insert records controlled by PAGE: rules to add data to a new partition (such as an off-page barcode partition).

Extracting Information from a Print Stream

To extract information from a print stream and create a flat file, follow these steps.

1. Create a control file. For more information, see **Developing a Control File** on page 75.
2. Create an Input tag group for each input print stream.
3. In the Input tag group, create a <FIELD> tag to identify the location of the data that you want to extract. You can define fields based on a fixed location on the page using X and Y coordinates

or you can define fields relative to other text on the page. For example, you could define a field for an account number that always appears after the text "Account Number: ".
4. Create an Output tag group.
5. In the Output tag group, create a Sidefile tag group. Side files are sometimes called data files or extract files. Each Sidefile tag group creates one data file that contains one record for each document processed in the specified output. You can use multiple Sidefile tag groups to create multiple side files.
6. In the Sidefile tag group, create a <FILE> tag to define the filename to use for the side file. For information see the *Enrichment Language Reference Guide*.
7. In the Sidefile tag group, create a <SIDEPART> tag to define each variable or constant to include in the side file. Enrichment assembles the data into records, one record per document. Use as many <SIDEPART> tags as necessary to define the record to write to the side file. For more information see the *Enrichment Language Reference Guide*.

Example

Here is an example control file that extracts information from a print stream and writes it to a side file.

```
<input>
   <name> EXM1
   <file> H:\PrintStreams\INPUT1.afp   <! Input file.                >
   <type> AFPL A                       <! File is AFP line data.     >
   <document> 1                        <! Every document is 1 page.  >
   <field> %%Customer_Number K         <! Find the customer number.  >
     <location> 8 60 8
   </field>
   <field> %%invoice   K               <! Find the invoice amount.   >
     <location> 9 60 10
   </field>
</input>
<output>
   <name> MEMO
   <file> H:\Output\output1.afp        <! Output print stream.       >
   <sidefile>
     <file> H:\Sidefiles\sidefile1.txt <! Report file.               >
     <sidepart> %%Customer_Number 8 R ' '
     <sidepart> %%invoice   11 R ' '
     <sidepart> %%DOCUMENT_NO 5 R 0    <! Sequential document number.>
   </sidefile>
</output>
```

Postal Processing

Enrichment can interface with external applications to perform the various types of postal processing.

CASS™ Processing

The Coding Accuracy Support System (CASS™) is a United States Postal Service® (USPS®) program that certifies the accuracy of address validation software. To qualify for certain postal discounts you must use software that is CASS Certified™ to assign ZIP Code™, ZIP + 4® codes, and delivery point barcodes to mail.

Makers of address validation software must pass a test designed by the USPS in order to have their software designated as CASS Certified. CASS Certified software must pass tests of accuracy in the following areas:

- Five-digit coding
- ZIP + 4/delivery point (DP) coding
- Carrier route coding
- Delivery Point Validation (DPV™)
- Locatable Address Conversion System (LACSLink™)
- Enhanced Line of Travel (eLOT™)
- Residential Delivery Indicator (RDI™)

When you use a CASS Certified product, you are assured of the following minimum levels of accuracy:

Table 13: CASS Certification Levels

Certification Level	Required Accuracy Level
ZIP	98.5%
Carrier Route	98.5%
ZIP+4	98.5%
Delivery Point Barcode	100%
eLOT	100%

Certification Level	Required Accuracy Level
Perfect Addresses	100%

Enrichment is not a CASS Certified product, but it can be used with CASS Certified software to standardize addresses in print streams to USPS® requirements. To do this, Enrichment extracts the addresses from the documents and passes them to a CASS Certified address cleansing program. Address cleansing programs verify addresses and return the complete postal codes used to build the IMB™ barcode.

You can use Finalist to standardize addresses.

CASS Processing - Specifying Address Information

Addresses are special variables used by Enrichment to perform CASS™ processing. There are two ways to identify an address:

- The Input group <ADDRESSBLOCK> tag
- The Address tag group within the Input group.

Because an address is associated with a specific input stream, these tags go within the Input tag group. Both methods are limited to addresses of up to six lines.

The Input group <ADDRESSBLOCK> tag is the easiest way to identify addresses. However, you can only use it if:

- The address lines are always in the same position on the printed page
- Each address line is always located one line or record below the preceding address line, in the same location on the record
- The address is on the first page of the document.

If the address location moves on the page or if each address line is not located within the print stream one below the other (which is common in composed AFPDS or AFP line data), then you must use the Address tag group. For information about using the Address tag group, see the *Enrichment Language Reference Guide*.

CASS Processing - Enrichment Address Variable Processing

Enrichment performs the following special processing on address variables when you use CASS™ processing:

- The address variables are passed to the CASS™ program. The resulting cleansed address is written into system variables called *%%CASS_ADDR1* through *%%CASS_ADDR6*. Enrichment also creates many other system variables. Refer to the *Enrichment Language Reference Guide* for more information on the system variables created during postal processing.

- The system variable *%%ADDRESSCHANGED* identifies if the address was changed by the CASS™ software. You can also check the system variables associated with the return code for the cleansing (such as *%%LPCRC*) to perform conditional processing.
- Unless the CASS group <FIELDREPLACE> tag is set to NO the values of *%%CASS_ADDR1* through *%%CASS_ADDR6* will replace the values of the address variables. If the <FIELD> tag action parameter is set to R (replace) for these fields, the document is updated. If you set the <FIELDREPLACE> tag to NO, you must manually set the values of the address variables in the rule file if you want the address updated in the print stream.

The address variables and system variables can be used just like any other variables in the rules.

Enabling CASS Processing

To use CASS™ processing, follow these steps:

1. Verify that you have properly configured your Enrichment environment to use Finalist. For configuration instructions, see the *Installation Guide*.
2. Create a control file. For more information, see **Developing a Control File** on page 75.
3. Create an Input tag group for the input print stream that contains the addresses you want to standardize.
4. If the address information is always in the same position on the printed page, use an Input group <ADDRESSBLOCK> tag to specify its location.

   Note: The <ADDRESSBLOCK> tag can be used only for line data print streams. You must use the Address tag group with composed streams, such as AFPDS, Metacode, PCL and PostScript.

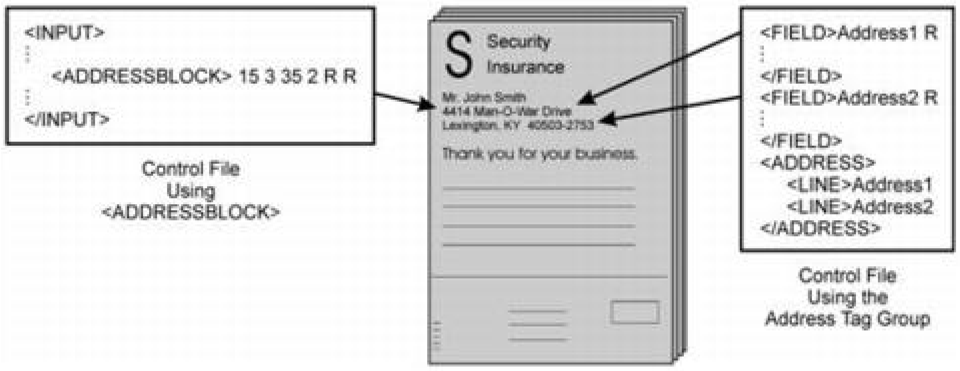
5. If you cannot use an <ADDRESSBLOCK> tag, do the following:

   a. In the Input tag group, use the <FIELD> tag to define each address line as a field. Specify the R parameter on the <FIELD> tag so that the address line can be replaced with the standardized address returned from the CASS™ program. For example:

   ```
   <FILED>Address1 R
   ```

   b. Use the Address tag group to identify the fields that comprise the address information to cleanse.

   The following illustration shows the difference between defining addresses with <ADDRESSBLOCK> and using <FIELD> tags in conjunction with the Address tag group.



6. In the Input tag group, specify the following to enable address cleansing for this input print stream:

   ```
   <CLEANSE>Y7
   ```

7. Create a CASS tag group.
8. In the CASS tag group, create a <CASSTYPE> tag to specify that you will use Finalist for CASS™ processing. For more information on the <CASSTYPE> tag, refer to the *Enrichment Language Reference Guide*.
9. If the input is not already in ZIP Code™ order, specify the <DOUBLESORT> tag in the CASS tag group. This tag allows you to improve performance by sorting the input into ZIP Code™ order before CASS™ processing. For more information see the *Enrichment Language Reference Guide*.
10. In the CASS tag group, create a <FIELDREPLACE> tag. This tag specifies whether or not to replace the address line or address block is to be replaced after the address is standardized. For more information see the *Enrichment Language Reference Guide*.
11. Add additional tags to the CASS tag group to control CASS™ processes as needed. For more information about the tags, refer to the discussion on the CASS tag group in the *Enrichment Language Reference Guide*.
12. (Optional) Place documents whose addresses were successfully standardized in one output and documents whose addresses could not be standardized in another.

    a. Create two Output tag groups, one for addresses that were successfully standardized and one for those that were not.
    b. Create a Rule tag group.
    c. In the Rule tag group, create the Content tag group.
    d. In the Content tag group, in the DOCUMENT section, write an If statement that separates documents that were successfully standardized from those that were not. The following example separates documents based on the return code from Finalist and assumes that there is an Output tag groups that defines an output named GoodOut and another Output tag group that defines an output named BadOut.

```
<rule>
   <content>
    DOCUMENT:           <!The rule section for document processing.>
    if %%LPCRC < 1 then <!Separate the documents based on the      >
```

```
       <output> GoodOut  <!return code from the CASS program.       >
     else
       <output> BadOut
     endif
    </content>
 </rule>
```

CASS Processing - Example with Finalist

The following is an example control file that illustrates how to perform CASS™ processing using Finalist.

```
<input>
   <name> CLEANSE              <! Identifiable name.        >
   <file> H:\Input\MyInput.afp <! Input file name.          >
   <type> AFPL A             <! AFP line data w/ ANSI controls.>
   <doc> T %%AcctNum CHANGE  <! Top of document when Account changes.>
   <field> %%AcctNum KA      <! Find all occurrences.      >
      <loc> 13 60 8          <! Line 13, column 60, for 8 bytes.  >
   </field>   <addressblock> 9 9 35 4 L R <! Define address block for
cleansing >
   <cleanse> YES               <! Run the CASS cleanse program.      >
</input>
<cass>
   <casstype> LPC          <! Specify Finalist as the CASS program.  >
   <doublesort> YES        <! Sort the documents in zip code order   >
                           <!   before calling CASS program. This    >
                           <!   improves performance. Next specify   >
                           <!   reason code threshold for each of the>
                           <!   nine Finalist reason                 >
                           <!   codes and one general return code.   >
   <lpcreplace> 1 5 9 6 6 6 4 4 4 2
</cass>
<rule>
   <content>
    DOCUMENT:            <! The rule section for document processing.>
    if %%LPCRC < 1 then  <! Separate the documents based on the      >
      <output> GoodOut   <!   return code from the CASS program.     >
    else
      <output> BadOut
    endif
   </content>
</rule>
<output>
   <name> GoodOut               <! Identifiable name.  >
   <file> H:\Output\GoodOut.AFP <! Output file name.   >
</output>
<output>
   <name> BadOut                  <! Identifiable name. >
```

```
    <file> H:\Output\BadOutput.AFP <! Output file name.  >
</output>
```

Postal Presort

Presorting means sorting the documents in a print stream to minimize postal costs. Presorting software, such as Mailstream Plus, identifies the optimal sort order and produces the required postal reports. However, presorting software cannot operate on complex print streams. Enrichment interfaces with presorting software to perform postal presort processing on print streams.

Note: Presort software such as Mailstream Plus is Presort Accuracy Validation and Evaluation (PAVE™) certified by the USPS®.

Note: When Enrichment performs presort processing, it operates in "all-at-a-time" mode. For more information, see **Processing Flow** on page 49.

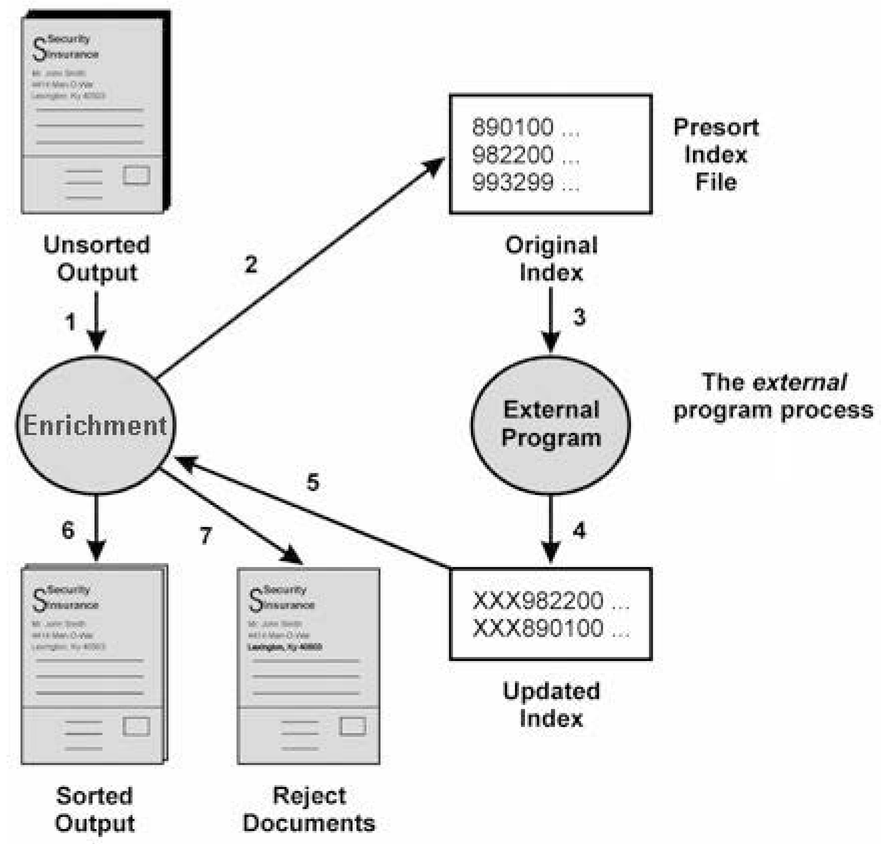
During presorting, Enrichment identifies key information on all documents and builds an index. The external presort program can sort the index file or remove records from it. Enrichment then reorders the print stream to match the sorted index file. If any records have been removed, Enrichment can write the documents associated with these records to a separate file (called a reject file).

Presorting effectively allows Enrichment users to change the order of documents in the print stream and optionally exclude certain documents. The external programs that Enrichment calls can be postal sorting programs or any other type of program that can manipulate an index.

As the following figure shows, Enrichment does the following during the presort process:

- Analyzes each document (**1**) in an output stream to gather the necessary sort information. Runs DOCUMENT: rules.
- Writes the sort information, one line per document, to the presort index file (**2**).
- After all documents have been analyzed, Enrichment calls the external program (**3**) to sort and purge records from the temporary index (**4**).
- Reads the sorted index (**5**) and reorders the documents to match the index order. Runs PRESORTED: rules and writes out the documents (**6**).
- Compares the original index to the sorted index to identify missing records.
- Optionally, writes all documents associated with missing records to a reject file (**7**).

Note: Enrichment performs these steps in a single run, thus simplifying operation and improving performance.



Use the Presort tag group within the Output group to specify outputs on which Enrichment is to perform a postal presort (or system sort). The Presort group must begin with the <PRESORT> tag and end with the </PRESORT> tag. Neither tag has parameters.

Using the Presort Tag Group to Call Other Programs

While originally designed to call postal presort software such as Mailstream Plus, the Presort tag group can be used to call virtually any program, including Windows batch files and UNIX shell scripts. Furthermore, the Presort tag group can consist of multiple external programs that operate on the presort index file. In this scenario, the <STEP> tag specifies each job step (that is, each program) to execute. Enrichment processes <STEP> tags in the order in which they occur in the Presort tag group. All of the programs specified by <STEP> tags run from within Enrichment, not as separate steps in your script.

Rule File Processing for Presort

Enrichment runs the PRESORTED: section of the rules for each document after presort. PRESORTED: rules are not run for outputs that are not presorted.

PRESORTED: rules are typically used to:

- Add banner pages between trays

- Break print streams between trays
- Create reports based on presorted documents.

For example, the following shows rule code that will compute and output a report of the number of customers and total pages by tray.

```
START:
   %%OLDTRAY =
    %%TRAYDOCS = 0
   %%TRAYPAGES = 0
PRESORTED:
   IF %%OLDTRAY <> %%TRAY_NO AND
      %%OLDTRAY <> THEN
      %%REPORT = %%TRAYDOCS | , | %%TRAYPAGES | , | %%OLDTRAY
      WRITE(DD:REPORT,%%REPORT)
      %%OLDTRAY = %%TRAY_NO
      %%TRAYDOCS = 0
      %%TRAYPAGES = 0
   ENDIF
   %%TRAYDOCS = %%TRAYDOCS + 1
   %%TRAYPAGES = %%TRAYPAGES + %%TOTAL_PAGES
FINISH:
   %%REPORT = %%TRAYDOCS | , | %%TRAYPAGES | , | %%OLDTRAY
   WRITE(DD:REPORT,%%REPORT)
```

Performing Presort

To perform postal presorting, follow these steps.

1. Verify that you have properly configured your Enrichment environment to use Mailstream Plus. For configuration instructions, see the *Installation Guide*.
2. Create a control file. For more information, see **Developing a Control File** on page 75.
3. Create Input tag groups for each input print stream. Identify the address on each input using the <ADDRESSBLOCK> tag or the Address tag group.
4. Create Output tag groups for each output print stream.
5. In the Output tag group, create the Presort tag group.
6. In the Presort tag group, create a <PRETYPE> tag to specify which program you will use to perform the postal presort (for example, Mailstream Plus).
7. In the Presort tag group, create a <FILE> tag to specify the presort index file that Enrichment will create.
8. In the Presort tag group, create a <SORTPART> tag and specify they system variable *%%DOCINDEX* as follows:

   ```
   <SORTPART> %%DOCINDEX 8
   ```

   This is required for presorting.

9. (Optional) In the Presort tag group, create additional <SORTPART> tags that specify additional information that you want to use to build the presort index file. For additional information, refer to the *Enrichment Language Reference Guide*.
10. In the Presort tag group, create a <STEP> tags to call your postal presort program (such as Mailstream Plus). For additional information, refer to the *Enrichment Language Reference Guide*.

Postal Presort Example

The following example demonstrates how to perform a postal presort on mainframe systems using a POSTNET™ Code barcode as the sort criterion.

```
<input>
   <name> INPUT                     <! Identifiable name.            >
   <file> DD:INPUT                  <! Input file name.              >
   <type> AFPL A                    <! AFP line data w/ANSI controls.>
   <doc> T %%AcctNum CHANGE  <! Top of document when field changes.  >
   <field> %%AcctNum KA           <! Find all occurrences            >
      <loc> 13 60 8           <! Line 13, column 60, for 8 bytes.    >
   </field>
   <field> %%SortBar K           <! The delivery point barcode field >
      <loc> 12 10 12          <! Print line 12, column 10, 12 bytes. >
   </field>
</input>
<rule>
   <content>
    <! Initialize the variables used in the rule.                 >
     START:
      %%Oldtray = ' '
      %%Traydocs = 0
      %%Traypages = 0
    <! Everytime there is a new tray, make an entry in the report >
    <! file that has the accumulated total documents pages.       >
     PRESORTED:
     if %%TRAY_NO <> ' ' THEN     <! don't increment for rejects  >
      if %%Oldtray <> %%TRAY_NO AND %%Oldtray <> '' then
         %%Report = %%Traydocs | ',' | %%Traypages | ',' | %%Oldtray
         WRITE('DD:TRAYRPT',%%Report,FB, 80, 8000)
         %%Oldtray = %%TRAY_NO
         %%Traydocs = 0
         %%Traypages = 0
      endif
      %%Traydocs = %%Traydocs + 1
      %%Traypages = %%Traypages + %%TOTAL_PAGES
     endif
    <! Make the last entry in the report for the last tray.         >
     FINISH:
      %%Report = %%Traydocs | ',' | %%Traypages | ',' | %%Oldtray
      WRITE('DD:TRAYRPT',%%Report,FB, 80, 8000)
   </content>
</rule>
<output>                                   <! The presorted output file. >
```

```
    <name> GoodOut                  <! Identifiable name.          >
    <file> DD:OUTPUT1               <! Output file name.           >
    <presort>                              <! Specify presort details. >
       <pretype> LPC               <! Type of presort to be done.     >
       <file> DD:MSSTIN
       <sortpart> %%DOCINDEX 8 L ' ' <! System variable, 8 bytes.     >
       <sortpart> %%SortBar 12 L ' ' <! POSTNET data, 12 bytes.       >
       <rejectfile> DD:REJECT   <! Any documents with invalid zipcodes>
       <! The presort steps to process with appropriate params and    >
       <! in the order of execution.                                   >
       <step> MSDR00
       <outfile1> DD:MSWKIJ4 400 <! Presorted, indexed file.          >
    </presort>
 </output>
```

ConnectRight Mailer

Enrichment can call ConnectRight Mailer (CRM) using ConnectRight Mailer's Hot Folder Monitor. For Enrichment to call ConnectRight Mailer, you must use <CASSTYPE> C and <PRETYPE> C.

The ConnectRight Mailer Import step must have a field layout beginning with:

- 16 bytes for an unmapped index
- 70 bytes for address line 1
- 70 bytes for address line 2
- 70 bytes for a City/State/Zip line
- 32 bytes for an Urbanization line

And it must end with a 2 byte field for the Carriage Return and Line Feed character

The ConnectRight Mailer Export step must have a field layout beginning with:

- 16 bytes for the unmapped index
- 70 bytes for address line 1
- 70 bytes for address line 2
- 43 bytes for the combined City/State/Zip line

You can include additional fields in either the Import or Export step.

<CRMHOTFOLDER>	The <CRMHOTFOLDER> tag must be included in the <CASS> group, describing where the CRM job will look for the input file.

<FILE>	In the <PRESORT> group, the <FILE> tag must be the CRM project name followed by .txt. For example: `<FILE> crmin.txt`
<SORTPART>	The first <SORTPART> tag must be: `<SORTPART> %%DOCINDEX 16 R ' '` The second <SORTPART> tag must be: `<SORTPART> %%SCAN_RESULTS 240 R ' '`
<OUTFILE1>	The <OUTFILE1> tag must point to the file name, excluding the path, of the Export step in ConnectRight Mailer, followed by the record length that ConnectRight Mailer will generate. Unless additional fields are included, the record length will be 201. For example: `<OUTFILE1> crmout.txt 201`
<INDEXCOL>	The <INDEXCOL> tag must be: `<INDEXCOL> 1 16`

By default, address lines will be replaced with the results from ConnectRight Mailer. If additional fields are included in the output, the CRM output record can be processed in the rule file. The system variable *%%PRESORT_RECORD* will contain the entire record from ConnectRight Mailer.

ConnectRight Mailer Example

An entire <CASS> group might look like this:

```
<cass>
 <casstype> C
 <crmhotfolder> C:\crmHotFolder\CRMIN
</cass>
```

And an entire <PRESORT> group might look like this:

```
<presort>
     <pretype> CRM
     <file> crmin.txt
     <sortpart> %%DOCINDEX 16 R ' '
     <sortpart> %%SCAN_RESULTS 240 R ' '
     <outfile1> crmout.txt 201
     <indexcol> 1 16
     <rejectfile> DDOUTPUT2 //resulted reject file is empty;
</presort>
```

Using Mail360 for Intelligent Mail Barcodes

Mail360 Manager assigns and manages Intelligent Mail Barcode® serial numbers. It provides the following functionality:

- Assignment and management of unique Intelligent Mail Barcode®
- Linking an Intelligent Mail® Barcode to an internally recognizable uniqueness identifier
- Integration of correspondence to improve business processes in departments such as billing, marketing, and customer support

While it is possible for you to use another tool to generate serial numbers for Intelligent Mail Barcodes, the Enrichment control file language provides elements that allow you to easily control Mail360 processing. If you do not have a license for Mail360 Manager, contact your sales representative.

To use the serial numbers generated by Mail360 Manager for barcodes applied by Enrichment, use the following:

`<MAIL360CONFIG>`	This tag specifies the location of the Mail360 configuration file. **Note:** For Enrichment to call Mail360 on z/OS, the XPLINK(ON) run-time option must be specified. For example: //M360JOB EXEC PGM=PDRSW000, PARM='XPLINK(ON)/'.
`IMBRange`	This function retrieves a range of consecutive serial numbers from Mail360 Manager.
`IMB`	This function encodes the IMB values into a string of digits representing the bars in the Intelligent Mail Barcode, which could be ascender, descender, full bar, or tracker.

See the *Enrichment Language Reference Guide* for complete information on these elements.

The following example control file shows how to utilize Mail360.

```
/*****************************************************************************/
/* This sample illustrates how to use Mail360 Manager to generate  */
/* Intelligent Mail Barcodes with Enrichment.                      */
/*****************************************************************************/
<environment>
   <mail360Config> C:\mail360\myg1imb.cfg  // Location of Mail360 config
 file
</environment>
/*****************************************************************************/
/* the ZIP Code is captured as a field. This field will be used  */
/* to build the Intelligent Mail Barcode.                        */
/*****************************************************************************/
<input>
```

```
   <name> InputFile
   <file> c:\jobs\Statements.afp
   <doc> 1
   <type> A
   <field> %%Zip KA
     <location> 1 78 5
   </field>
</input>

/**********************************************************************/
/* Static variables are set in the start section of the rule file  */
/**********************************************************************/
<rule>
   <content>
   start:
       %%Mailer_ID = "060540" // USPS-assigned mailer ID, 6 or 9 digits

       %%Barcode_ID = 00      // Barcode identifier; see USPS docs for
IMB
       %%Service_Type = 040   // Service type identifier; see USPS docs
 for IMB

       %%Partition_ID = "02"  // Mail360 partition ID; see Mail360 docs

       %%Block_Size = 500     // Initial number of serial nos to get
from Mail360

       /* IMBRange control variables used in mathematical computations
*/

       %%Remainder  = 0
       %%Serial_Num = 0
       %%Num_Avail  = 0

       %%RangeRC = 0
       %%RangeRV = 0
       %%RangeRM = " "

     document:
        /*
        * Get the serial numbers from Mail360 using IMBRange.
        * It returns the starting number and remainder.
        * %%Serial_Num will contain the starting serial number.
        * %%RV will contain the remainder if the full request
        * could not be fulfilled.
        */
/**********************************************************************/
/* 1) Check to see if there are serial numbers available and        */
/*    if it is 0 then request a block of new serial numbers         */
/* 2) Check the return codes and return values %%RC, %%RV, %%RM     */
/* 3) Upon successful return, assign serial number to a variable %%  */
/* 4) Subtract from the remainder, and keep looping to              */
/*      subtract numbers from the block.                            */
/* 5) If there is an error, write out a message.                          */
/*************************************************************************/

       if %%Num_Avail = 0 then
         %%Serial_Num = IMBRANGE(%%Mailer_ID, %%Block_Size,
                                  %%Partition_ID)

          %%RangeRC = %%RC
          %%RangeRV = %%RV
          %%RangeRM = %%RM
          if %%RangeRC = 0 then
             %%Remainder = %%RangeRV
             %%Num_Avail = %%Block_Size - %%Remainder
             %%Block_Size = 3      // For subsequent GetRange requests
          else
             %%Msg = "Error code = " | %%RangeRC | " %%RangeRM=" |
                      %%RangeRM | "%%RangeRV=" | %%RangeRV
             WRITE("ErrorLog.txt", %%Msg)
          endif
       endif

      /*
       *  Generate the IMB
       */

      // Fix length of serial number.
      // Mail360 Manager always returns 9 bytes.
      // A 6 digit Mailer ID results in a 9 digit serial number
      // A 9 digit Mailer ID results in a 6 digit serial number

       if LENGTH(%%Mailer_ID) = 6 then
          %%Serial_Num = JUSTIFY(%%Serial_Num, R, 9, 0)
       endif
       if LENGTH(%%Mailer_ID) = 9 then
          %%Serial_Num = JUSTIFY(%%Serial_Num, R, 6, 0)
       endif

       // concatenate all required fields into a %%payload variable
      // call the IMB function and the string to be encoded is returned

       // into the %%NUMS variable
      // increment %%Serial_Num counter & decrement %%Num_Avail counter


       %%payload = %%Barcode_ID | %%Service_Type | %%Mailer_ID |
%%Serial_Num | %%Zip
       %%NUMS = IMB(%%payload)
       %%Serial_Num = %%Serial_Num + 1
       %%Num_Avail = %%Num_Avail - 1

   </content>
</rule>
```

```
/***********************************************************************************/
/* Add the IMB barcode using the %%NUMS variable data                            */
/***********************************************************************************/

 <add>
     <addtype> IMB
     <addpart> %%NUMS 65
     <position> 1 10
 </add>
/***********************************************************************************/
/* Write the output file                                         */
/***********************************************************************************/

 <output>
   <name> OutputFile
   <file> c:\jobs\StatementsWithIMB.afp
</output>
```

Using IMpb barcodes with <CODE128>

This sample illustrates how to encode an Intelligent Mail package barcode (IMpb), including the required FNC1 character. To generate the FNC1 character in an Enrichment-generated Code 128 barcode, you must use x'16'. Different IMpb formats may require the FNC1 character in fewer, or different, locations.

For more information about:

- Code 128 barcodes, see the *Enrichment Language Reference Guide*
- IMpb, go to **https://ribbs.usps.gov/intelligentmail_package/documents/tech_guides/PUB199IMPBImpGuide.pdf**.

```
/***********************************************************************************/
/* This sample illustrates how to use the <CODE128> tag to generate */
/* Intelligent Mail package barcodes with Enrichment.               */
/***********************************************************************************/
<INPUT>
  <NAME>INPUT
  <file>C:\test\blankpage.afp
    <TYPE>A
    <density> 240
</INPUT>
<OUTPUT>
  <NAME>OUTPUT
  <add>
  <addtype>  128
  <code128> C
  <addpart> x'16'       // FNC1
  <addpart> 420         // Routing Application ID
```

```
   <addpart> 123456789   // Zip Code (9 digits)
   <addpart> x'16'       // FNC 1
   <addpart> 94          // Channel Application ID
   <addpart> 123         // Service Type Code
   <addpart> 12          // Source ID
   <addpart> 912345678   // Mailer ID
   <addpart> 1234567     // Serial Number
   <addpart> 1           // MOD 10 check digit
   <bars> 3 PELS
   <height> 60 PELS
   <orient> 1
   <position> 100.0 1000.0 PELS
   </add>

    <FILE>impb.afp

  </OUTPUT>
```

The generated barcode for this sample is:

Managing Print Stream Resources

Print stream resources are items such as images and fonts that are used in AFPDS and PCL print streams. For more information on resource management, see the following topics.

Merging Resources

Resource merging is useful when splitting, sorting, or merging files. For example, if Enrichment is merging two input files that both have font FONT1234 as a resource, Enrichment can identify which instance of FONT1234 to use in the merged output. If Enrichment determines that both fonts are the same, the first font is used and the second one is discarded.

To enable resource merging, use these Enrichment language elements:

<RESOURCESCAN>	This tag enables resource management. Enrichment scans each input record, reconciling resources with duplicate names and creating new unique names if resources are different.

<RESOURCEREPLACE>	This tag specifies whether or not Enrichment should remove and/or replace inline resources for a specific output file.
%%NUM_FONTS	This system variable indicates the number of fonts on a page.
GETFONT	This function returns the AFPDS font whose local ID in the MCF record for the page matches the requested font number. This function can only be used in the PAGE section of the rule file.

For complete information on <RESOURCEREPLACE>, <RESOURCESCAN>, %%NUM_FONTS, and GETFONT, see the *Enrichment Language Reference Guide*.

Using Resource Files

Enrichment can extract resources from a print stream and create separate resource files for each resource, or a single file for all resources in a given input. These resource files can then be read in to other input print streams, allowing you to apply resources consistently across multiple inputs. For example, Enrichment can create a font resource file for each font it finds. These font files can then be used by other input print streams to ensure the consistent use of fonts.

To create and use resource files, use the following tags:

<RESOURCEOUTFILE>	This tag enables the creation of resource files. Use it to specify the file, directory, or PDS (on mainframe systems) where you want Enrichment to create the resource files.
<RESOURCEEXTENSION>	On Windows, Linux, and Unix, this tag specifies the file extension to use for each type of resource file (font resources, overlays, images, etc.). This tag is not used on mainframe systems.
<RESOURCEINFILE>	This tag specifies a resource to read into a print stream. If this tag is present, <RESOURCESCAN> is set automatically set to YES.

For complete information on these tags, see the *Enrichment Language Reference Guide*.

Logging Resource Activities

Enrichment can optionally log the names of the resources found in the input print stream. To enable resource logging, use the <RESOURCELOGFILE> tag. For complete information on the use of this tag, see the *Enrichment Language Reference Guide*.

Creating a Vault Journal File

Enrichment can create a Vault journal file for the print streams it processes, allowing you to archive Enrichment output in Vault.

> **Note:** If you have licensed Visual Engineer, there is a utility available that greatly simplifies the process of generating a Vault journal file. For more information, see the *Visual Engineers User Guide*.

To create a Vault journal file:

1. In your Enrichment control file, add the following code. This code creates <XMLELEMENT> tag groups to map the XML elements to user defined variables. The variables are populated by rule file processing, which will be added in a later step.

```
<xmlelement>
  <xmlname> endeGAD
  <content>
    </eGAD>
  </content>
</xmlelement>

<xmlelement>
  <xmlname> jobdata
  <content>
    <?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <!DOCTYPE eGAD SYSTEM "eGAD.Dtd">
    <eGAD pakUID="%%DTD_pakUID" >
    <jobdata>
      <datetime>%%DTD_datetime</datetime>
      <platform>%%DTD_platform</platform>
      <Version major="%%DTD_Version_major"
        minor="%%DTD_Version_minor" >
        %%DTD_Version</Version>
      <JobGUID>%%JOBGUID</JobGUID>
      <JobName>%%DTD_JobName</JobName>
      <JobShortName>%%DTD_JobShortName</JobShortName>
      <NativeFormat value="%%DTD_NativeFormat_value" />
      <!--repeatable-->
      <ResourceGUID value="%%DTD_ResourceGUID_value"
         p="%%DTD_ResourceGUID_p" />
      <!--/repeatable-->
    </jobdata>
  </content>
</xmlelement>

<xmlelement>
  <xmlname> document
  <content>
    <document docID="%%DTD_docID"
      docMasterID="%%DTD_docMasterID"
      docInstanceID="%%DOCGUID" >
      <VendorId>%%DTD_VendorId</VendorId>
      <DocTypeId>%%DTD_DocTypeId</DocTypeId>
      <AccNo>%%DTD_AccNo</AccNo>
      <StmtDate>%%DTD_StmtDate</StmtDate>
      <PayDetails>
        <DueDate>%%DTD_PayDetails_DueDate</DueDate>
        <!--repeatable-->
        <Amt type="%%DTD_PayDetails_Amt_type"
          curr="%%DTD_PayDetails_Amt_curr" >
          <Due>%%DTD_PayDetails_Amt_Due</Due>
          <MinDue>%%DTD_PayDetails_Amt_MinDue</MinDue>
        </Amt>
        <!--/repeatable-->
      </PayDetails>
      <!--repeatable-->
      <DDSDocValue name="%%DTD_DDSDocValue_name"
        type="%%DTD_DDSDocValue_type"
        len="%%DTD_DDSDocValue_len"
        UsedBy="%%DTD_DDSDocValue_UsedBy" >
      %%DTD_DDSDocValue</DDSDocValue>
    <!--/repeatable-->
    <CustData>
      <Name>%%DTD_CustData_Name</Name>
      <!--repeatable-->
      <Addr line="%%DTD_CustData_Addr_line" >
        %%DTD_CustData_Addr</Addr>
      <!--/repeatable-->
      <City>%%DTD_CustData_City</City>
      <Region>%%DTD_CustData_Region</Region>
      <PostalCode>%%DTD_CustData_PostalCode
        </PostalCode>
      <Country>%%DTD_CustData_Country</Country>
      <Phone>%%DTD_CustData_Phone</Phone>
    </CustData>
    <NumberOfPages value="%%DTD_NumberOfPages_value" />
    <Skipped>
      <!--repeatable-->
      <SPages>%%DTD_Skipped_SPages</SPages>
      <!--/repeatable-->
    </Skipped>
    <ToC>
      <!--repeatable-->
      <BMark level="%%DTD_ToC_BMark_level"
        name="%%DTD_ToC_BMark_name"
        pageID="%%DTD_ToC_BMark_pageID"
        lref="%%DTD_ToC_BMark_lref" >
      </BMark>
      <!--/repeatable-->
    </ToC>
  </document>
  </content>
</xmlelement>
```

2. Create an <OUTPUT> tag group. In the <OUTPUT> tag group:
   - Create an <XMLFILE> tag and specify a file name for the journal file.
   - Specify one or more <XMLPART> tags. The <XMLPART> tag indicates which XML elements to include in the journal file. (For more information, see the *Enrichment Language Reference Guide*.) The name you use in the <XMLPART> tag corresponds to the names you assigned to the XML elements with the <XMLNAME> tag.

   For example, the completed <OUTPUT> tag group may look similar to this:

```
<output>
  <name> Output1
  <file> c:\myfiles\statements.afp
  <xmlfile> c:\myfiles\journalfile.txt
  <xmlPart> jobdata H
  <xmlPart> document D
  <xmlPart> endeGAD T
</output>
```

3. In the rule file, assign values the variables defined in the <XMLELEMENT> tag groups. This will determine the values placed in the journal file's fields. For example:

```
<rule>
  <content>
    DOCUMENT:
    %%DTD_CustData_Addr[0] = %%Name
    %%DTD_CustData_Addr[1] = %%AddressLine1
    %%DTD_CustData_Addr[2] = %%AddressLine2
    %%DTD_CustData_Addr[3] = %%AddressLine3
  </content>
</rule>
```

Note: If a variable is defined in the <XMLELEMENT> tag group but not assigned a value in the rule file, the variable name is used as the value. For example, the variable *%%DTD_datetime* contains the value for the journal file tag <datetime> as specified in the <XMLELEMENT> tag group. If the variable *%%DTD_datetime* is not assigned a value in the rule file, then the variable name will be used as the field value in the journal file:

```
<datetime>%%DTD_datetime</datetime>
```

5 - Running an Application

In this section

Running on Mainframe Systems

If you are using a mainframe system, JCL controls Enrichment applications. On systems using modules built with an IBM compiler, specify run-time arguments in the Enrichment JCL PARM statement as follows:

```
PARM='Cruntime//switch1 /switch2'
```

where

Cruntime	Is any number of z/OS or C/370 run-time options to use during processing. Often, there are no C/370 run-time options specified. Do not use this parameter if you are operating a version of Enrichment compiled with SAS/C. If this parameter is applied, it must be separated from the arguments that follow with a concluding slash (/).
switchN	Is an Enrichment run-time argument. Each Enrichment switch must begin with a slash (/). For the first switch, this slash comes after the closing slash for the C/370 run-time arguments.

For a complete listing of run-time arguments, see **Run-Time Arguments** on page 208.

> **Note:** If you receive the error "CEE3191E An attempt was made to initialize an AMODE24 application without using the ALL31(OFF) and STACK(,,BELOW)", change your EXEC PDRSW000 statement to include PARM='ALL31(OFF),STACK(,,BELOW)/'.

A sample Enrichment JCL is shown below.

```
//*your job card
//**************************************************************
//** This is a sample Enrichment job.
//** Change SYS3.C370 to the high level qualifier for C/370 run-time.
//** Change SYS3.PLI to the high level qualifier for the PLI library.
//** Change PDR.STREAMW to the high level qualifier for Enrichment.
//** Change all DDs to reference your particular data set names.
//**************************************************************
//JOBLIB DD DSN=PDR.STREAMW.LOAD,DISP=SHR
// DD DSN=SYS3.C370.SEDCLINK,DISP=SHR
// DD DSN=SYS3.PLI.SIBMLINK,DISP=SHR
//*
//**************************************************************
//* This step runs Enrichment.
//SWEAVE EXEC PGM=PDRSW000
//* The control file can be either in-stream data or
//* a separate data set as the example shows.
//CONTROL DD DSN=CONTROL.FILE,DISP=SHR
```

```
//*
//* Input print stream for Enrichment job.
//INPUT1 DD DSN=INPUT.DATA,DISP=SHR
//*
//* You could optionally have a rule file.
//RULE DD DSN=OPTIONAL.RULE.FILE,DISP=SHR
//*
//* Output: Enhanced print stream
//OUTPUT1 DD DSN=OUTPUT.FILE,DISP=SHR
//*
//* Summary Report and Messages
//REPORT DD DSN=MESSAGE.SUMMARY,DISP=SHR
//
```

Temporary File Storage

When used to sort, presort, double-sort, or match documents, Enrichment copies entire input print streams to temporary storage to enable random retrieval after the sort. Enrichment uses extended memory, Hiperspace memory, and one or more entry-sequenced data set (ESDS) VSAM files for temporary storage.

Note: Enrichment only caches input print streams if you are sorting, presorting, or double-sorting documents. In all other configurations, Enrichment processes documents one at a time and does not use Hiperspace, VSAM, or disk.

To reduce disk I/O, Enrichment uses memory (above the 16MB) to temporarily store inputs for processing. If Enrichment exceeds a user-specified amount of memory, it then uses Hiperspace memory to hold the inputs, when available. If Hiperspace is not available, is exceeded, or if a user-specified amount of Hiperspace is exceeded, Enrichment uses ESDS VSAM files for temporary storage. In many cases, you will not need to use VSAM or disk storage with Enrichment. However, if an input is very large, you can specify that it be stored exclusively in Hiperspace or VSAM or disk. Set the <IO> tag to HIPERSPACE to use Hiperspace exclusively, to DISK to use VSAM or disk exclusively, or to TRANSFER to use Hiperspace then VSAM.

Note: Make sure to set the <SINGLEBUFFER> tag when doing large document processing. <SINGLEBUFFER> allows you to define the initial size and growth factor for the single document buffer.

Use the Input group <TEMPDISK> tag to identify VSAM or disk files to Enrichment. If it is possible that your input data could exceed the capacity of a single ESDS VSAM file, you can use multiple <TEMPDISK> tags to specify multiple files.

Insert pages can be stored in memory, Hiperspace, or disk. Or, if uniquely set for each document, the insert pages may not be stored at all by using <IO> NONE.

Using Hiperspace for Temporary File Storage

If you do not have sufficient memory to store documents temporarily, you may want to use hiperspace memory. Three tags affect how Enrichment uses hiperspace: the Input group <IO> tag, the Environment group <HIPERMAX> tag, and the Environment group <HIPERTRANSFER> tag.

Note: Hiperspace is only available on mainframe systems when you use the IBM C run-time libraries.

<HIPERMAX> Tag

If you set <IO> to TRANSFER or OPTIMUM, the Environment group <HIPERMAX> tag specifies the maximum amount of hiperspace to use before switching to disk. The valid range for <HIPERMAX> is 1K to 2048M.

Note: If your application uses all available hiperspace, Enrichment will abend when it tries to transfer to disk. To avoid this, set <HIPERMAX> to an amount less than the maximum available hiperspace.

<HIPERTRANSFER> Tag

The Environment group <HIPERTRANSFER> tag defines the size of a memory area used by Enrichment to send data to and from hiperspace. The transfer area is only used if hiperspace is used. The valid range for <HIPERTRANSFER> is 0K to 2M.

Note: If Enrichment issues a PDR2164 or PDR2165 error, it is probably caused by a hiperspace environment issue. Lower your <HIPERTRANSFER> setting until Enrichment processes without issuing the error. If the error persists, set <HIPERTRANSFER> to 0 so Enrichment only uses memory and disk for storage.

C run-time defaults to a transfer area size of 16K. Enrichment's default for <HIPERTRANSFER> is 100K. We recommend that you set <HIPERTRANSFER> no lower than 16K, unless you set <IO> to OPTIMUM and want Enrichment to go from memory straight to disk. In this case, as above, set <HIPERTRANSFER> to 0.

Using VSAM Files for Temporary File Storage

If you prefer not to use memory for temporary document storage, you can use VSAM files. Using VSAM files reduces performance but saves memory. The VSAM files must be reusable and must be large enough to contain the entire input print stream or print streams being processed.

The VSAM files must exist before Enrichment can run. You do not need to load the ESDS VSAM files prior to using Enrichment because Enrichment performs the first load if necessary. However, Enrichment cannot create the files.

Each time Enrichment uses VSAM files for temporary data, it destroys the previous contents of the files. Therefore, you can use the same VSAM files for a production Enrichment application, even if you run the application daily. However, you should use different VSAM files for different Enrichment applications.

To use VSAM for temporary document storage:

1. In the control file, set the Input tag group <IO> tag to DISK.

   Note: You can also use the <IO> tag in the Banner, Insertpage, and Insertrec groups to define banner or insert storage.

2. In the appropriate Banner, Input, Insertpage, or Insertrec tag group, use the <TEMPDISK> tag to identify the VSAM file that you want to use for temporary document storage.

   If you do not specify the <TEMPDISK> tag, the default DD:WORKVSAM is used and you must ensure that the data set exists or Enrichment processing will stop. You must define this file as reusable, and it must be large enough to contain the entire input print stream or print streams being processed.

   Note: When using <TEMPDISK> to define VSAM for AFPDS, the maximum record size must be at least 8205 or data records may get truncated.

3. If the input is so large that the VSAM storage requirement exceeds a disk pack, you can use multiple VSAM files (none to exceed a pack). When setting this up, you must specify one <TEMPDISK> tag in the Enrichment control file for each additional VSAM file.
4. Allocate a VSAM file using the reserved DD name WORKVSAM in your JCL to point to the VSAM file. The figure below shows an example of JCL statements used to create an ESDS VSAM file (sometimes called a cluster) for use with Enrichment.

```
//*********************************************************************
//* Example of how to define an ESDS VSAM data set
//*
//* To use this job:
//* 1) Supply a proper job card
//* 2) change the following IDCAMs statement values...
//* datasetname -> the name you want for your data set
//* size -> the number of cylinders to allocate
//* avglrecl -> the average record size of your input
//* maxlrecl -> the maximum record size of your input
//* volume -> the volume to allocate the data set on
//*********************************************************************
//IDCAMS EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=N
//SYSIN DD *
DELETE                                      -
datasetname                                 -
CLUSTER
DEFINE CLUSTER                              -
(NAME (datasetname)                         -
 CYLINDERS (size) RECSZ (avglrecl maxlrecl) -
 VOL (volume) REUSE                         -
 NONINDEXED                                 -
)
```

Blocking Input and Output Data Sets on Mainframe Systems

File blocking can significantly impact Enrichment performance on mainframe systems. Enrichment is written in C and requires special file blocking for optimum performance. While COBOL buffers block themselves and therefore control their own performance, C requires you to set the blocking in the JCL (for example, DCB=(BUFNO=30)). Keep the following in mind when you set blocking:

- Always use 1/2 track blocking, even when routing output to SYSOUT.
- Try to buffer 1/2 cylinder (BUFNO=15). If possible, full cylinder buffering is recommended (BUFNO=30).
- Use BUFNO for both inputs and outputs and to TAPE as well as DASD.
- Use IEBGENR to reblock bad inputs to a temporary file before running Enrichment (for example, when LRECL and BLKSIZE are both 133). This can improve I/O performance.

These I/O buffers are obtained from memory below the 16MB line, so if you have many inputs or outputs you can run out of memory for Finalist or other programs that run below the line. If you cannot use BUFNO=15 because of memory problems, only use BUFNO on your largest inputs and outputs or drop the BUFNO to 10 or 5.

The following shows JCL in which blocking is properly set.

```
//INVOICES DD DSN=PDRC.SW.DATA,
//            DISP=(NEW,CATLG,DELETE),
//            DCB=(LRECL=133,BLKSIZE=27930,RECFM=FBA,BUFNO=30),
//            SPACE=(CYL,(10,10),RLSE),UNIT=SYSDA
```

Running on UNIX

You can run Enrichment in scripts, from the command line, from within your programs, or in any other manner that you would use a standard UNIX application.

The easiest way to run Enrichment is through the `../Enrichment/bin/swvrexe` script. With `swvrexe`, you pass Enrichment arguments on the `swvrexe` command line just as you would if you were executing Enrichment directly. The script does the following:

- Sets up PATH to point to Enrichment
- Establishes SWVR_LIB

- Establishes PBSSTTY
- Sets up a library path to point to callable library functions

Specify run-time arguments using this syntax:

```
sweaver -Arg1 -Arg2 -Arg3
```

where *ArgN* is an Enrichment run-time argument. Each argument must be separated from the preceding argument by a space and must begin with a dash (-). For a complete listing of run-time arguments, see **Run-Time Arguments** on page 208.

If you do not use `swvrexe` to run Enrichment, you must add the directory that contains the Enrichment executable to your path so that you can run it directly from the command line.

> **Note:** AIX® uses a delayed paging slot allocation technique for storage allocated to applications. When storage is allocated to an application with a subroutine, such as `malloc`, no paging space is assigned to that storage until the storage is referenced. This technique is useful for applications that allocate large sparse memory segments. However, this technique can affect portability of applications that allocate very large amounts of memory. If the application expects that calls to `malloc` will fail when there is not enough backing storage to support the memory request, the application might allocate too much memory. When this memory is referenced later, the machine quickly runs out of paging space and the operating system kills processes so that the system is not completely exhausted of virtual memory. The application that allocates memory must ensure that backing storage exists for the storage being allocated. Setting the PSALLOC environment variable to `PSALLOC=early` changes the paging space allocation technique to an early allocation algorithm. In early allocation, paging space is assigned once the memory is requested.

For more information, refer to “Paging space and virtual memory in Operating system and device management” from your AIX manual for more information.

Enrichment Executables

Enrichment has two UNIX executables that you use in different processing scenarios.

sweaver

Use this executable if you do not need to process input and/or output files larger than 2 GB. An example usage is:

```
sweaver -t=f -c=control.file -m=message.summary
```

sweaverL

Use this executable to process input and/or output files larger than 2 GB. This executable has higher system requirements than `sweaver`. See the *Installation Guide* for the system requirements for processing large files. An example usage is:

```
sweaverL -t=f -c=control.file -m=message.summary
```

Temporary File Storage on UNIX

When used to sort, presort, double-sort, or match documents, Enrichment copies entire input print streams to temporary storage to enable random retrieval after the sort. On UNIX systems, Enrichment uses virtual memory and disk files for temporary storage. Enrichment can automatically switch between storage methods or you can specify the type to use by using the <IO> tag.

Running Simultaneous Jobs on UNIX

Running multiple Enrichment jobs simultaneously on a UNIX system reduces performance by a factor equal to the number of jobs you are running. For example, if you run two Enrichment jobs simultaneously, performance is reduced by approximately 50 percent. You can improve performance for running multiple jobs by increasing the amount of RAM on the system. Another way to improve multi-job performance is to run one Enrichment process on one fixed disk drive and the other Enrichment process on a second fixed disk drive. Performance can further be improved if you have separate controllers, one for each fixed disk drive doing I/O.

Running on Windows

Enrichment runs like any other Windows application. You should add the directory that contains the Enrichment executable to your workstation's PATH environment variable so that you can run it directly from the command line. You can also use Enrichment in scripts, from the command line, from within your programs, or in any other manner that you would use a standard Windows application.

To run the executable from the command prompt use this syntax:

```
sweaver -c=control.file -m=message.file
```

Specify run-time arguments using this syntax:

```
sweaver -Arg1 -Arg2 -Arg3
```

where *ArgN* is an Enrichment run-time argument. Each argument must be separated from the preceding argument by a space and must begin with a dash (-). For a complete listing of run-time arguments, see **Run-Time Arguments** on page 208.

Temporary File Storage on Windows

When used to sort, presort, double-sort, or match documents, Enrichment copies entire input print streams to temporary storage to enable random retrieval after the sort. On Windows systems, Enrichment uses virtual memory and disk files for temporary storage. Enrichment can automatically switch between storage methods or you can specify the type to use by using the <IO> tag.

Run-Time Arguments

You can provide arguments to Enrichment at run time to supplement or override certain tag values specified in the control file and to create user-defined variables. The following table shows valid run-time arguments:

Argument	Description
-#=jobid	Specifies a job number to use for control files using a transform work flow. The default is 1.
-A=*traceLevel,[n]*	Specifies whether to enable tracing for Finalist processing. This can be used to isolate problematic addresses during the cleansing process. *traceLevel* is one of the following: A trace level value where: • 1—Displays ADDRSCAN input. • 2—Displays ADDRSCAN results. • 4—Displays Finalist input. • 8—Displays Finalist results. A combination of these values can be used. For example: A=12 Enable 4 and 8 (Finalist input and output). A=7 Enable 1, 2 and 4 A=15 Enable all A=3 Enable 1 and 2 If an invalid value is specified Enrichment uses the default value of 15. *n* specifies the number of documents to process prior to enabling the trace. This is optional

Argument	Description	
-C=*filename*	Specifies the control file to use for Enrichment processing. If you do not use this switch, Enrichment assumes the control file is DD:CONTROL for mainframe systems or STREAMW.CON for UNIX and Windows. If you have not defined DD:CONTROL on your mainframe system, processing halts.	
-E=*errorlevel*	Defines the level of error at which Enrichment processing will stop, where *errorlevel* is one of the following:	
	S	Stops processing only if Enrichment issues a severe message.
	W	Stops processing if Enrichment issues a warning or severe message.
	I	Stops processing when Enrichment issues an informational, warning, or severe message.
	This switch overrides the Environment tag group's <ERRORLEVEL> tag setting in the control file. If you do not use this switch, Enrichment stops processing according to the <ERRORLEVEL> tag setting. If you have not set the <ERRORLEVEL> tag, processing stops only when a severe message occurs.	
-F=*filename*	Specifies the name of the file or DD in which the compiled or decompiled control file will be stored. The default compile DD is DD:COMPCNTL for mainframe systems or STREAMW.CMP for UNIX and Windows. The default decompile DD is DD:DECNTL for mainframe systems or STREAMW.DMP for UNIX and Windows.	
-G=*filename*	Specifies the name of the file or DD in which the memory log will be stored. This will store all memory allocations and deallocations used by Enrichment. Do not use this argument unless instructed to by Technical Support. There is no default.	
-I=*include*	Indicates additional content that you want to include in the Enrichment report file. Currently the only valid value is R, which specifies the inclusion of a numbered listing of the rule file in the Enrichment report. Encrypted rule files cannot be included in the Enrichment report.	
-K=*key*	Identifies a key used to compile or decompile control files. A key consists of up to 10 characters that are used to lock or unlock the control file. Any rules included in the control file with the Content tag group are also compiled or decompiled using the key. The first time a control file is processed using a specific key, it is compiled. The next time it is run using the same key, it is decompiled. For more information on compiling and decompiling, refer to **Securing a Control File** on page 77.	

Argument	Description
-L=*length*	Specifies the size of Enrichment output messages, where length is one of the following: S — Messages break to the next line every 80 characters. L — Messages break to the next line every 132 characters. This switch overrides the Environment tag group <REPORTLENGTH> tag setting in the control file. If you do not use this switch, Enrichment breaks messages according to the <REPORTLENGTH> tag setting. If you have not set the <REPORTLENGTH> tag, Enrichment breaks messages to the next line every 80 characters. This switch overrides the Environment tag group <REPORTLENGTH> tag setting in the control file. If you do not use this switch, Enrichment breaks messages according to the <REPORTLENGTH> tag setting. If you have not set the <REPORTLENGTH> tag, Enrichment breaks messages to the next line every 80 characters.
-M=*filename*	Specifies the filename for the Enrichment message file. If you do not use this switch, Enrichment assumes the message file is DD:REPORT for mainframe systems or STREAMW.MSG for UNIX or Windows. If you have not defined DD:REPORT on your mainframe system, Enrichment routes messages to JES output (as if you specified SYSOUT=*) and generates a warning.
-O=*filename*	Specifies the filename for the Enrichment uplift report. The uplift report is generated whenever you use a secondary CASS™ product. (To use a secondary CASS™ product, specify the second parameter of the <CASSTYPE> tag.) The Enrichment uplift report shows you the CASS™ coding percentage for each CASS™ product and how much the secondary CASS™ product improved your coding percentage. The Enrichment uplift report is generated whenever you use a secondary CASS™ product, regardless of whether or not you specify the O run-time parameter. If you do not specify the O parameter the uplift report is placed in the default location as follows: • Windows and UNIX: The uplift report is placed in the working directory as is named uplift.txt. • Mainframe: The uplift report is placed in DD:UPLIFT.

Argument	Description	
-R=*print*	Specifies which messages Enrichment should include in the message file, where print is one of the following:	
	N	Reports no messages.
	D	Reports detailed messages. This is the most comprehensive level of reporting.
	S	Reports only severe messages. Severe messages reflect conditions that halt Enrichment processing.
	W	Reports warnings and severe messages. Warning messages reflect error conditions that may not cause Enrichment to stop but can produce unexpected processing results.
	I	Reports informational, warning, and severe messages. Enrichment displays informational messages to indicate progress during processing. Informational messages require no corrective action.
	P	Reports processing, informational, warning, and severe messages. Processing messages are more specific than informational messages and may be helpful as troubleshooting aids.
	This switch overrides the Environment tag group <MESSAGELEVEL> tag in the control file. If you do not apply this switch, Enrichment uses the <MESSAGELEVEL> tag setting to place messages in the processing report. If you have not set the <MESSAGELEVEL> tag, Enrichment places processing, informational, warning, and severe messages in the report.	
-S=*messagenumYN*	Defines whether Enrichment will place message numbers in the reports it generates, where *messagenumYN* is one of the following:	
	Y	Message numbers appear in the report.
	N	Message numbers do not appear in the report.
	If you do not use this switch, Enrichment assumes S=Y.	

Argument	Description
-T=*tracelevel*	Specifies the level of trace information to include in the Enrichment Report, where *tracelevel* is one of the following: I — Includes an intermediate amount of trace information. F — Includes full trace information. This switch is useful for troubleshooting, especially if there appear to be problems with Enrichment output.
-U=*varname*=*value*	Creates a user-defined variable, *%%varname*, and assigns to it the given value. For example, `-U=myvar=some_value` This would create a variable named `%%myvar` with a value of `some_value`. You can define as many unique variables as you want but you can only specify each variable once. Do not include the double percent (%%) when you define the variable name. Enrichment will automatically add this. The variable name cannot be a system, CASS, or presort variable. The maximum total length of the variable name, second equal sign, and value is 129 characters. Command line variables may be used as file names on all file tags, including <REJECTFILE>. Using command line variables for file names does not affect the behavior of DYNAFILE. **Note:** The rule file may change the value of a command line variable used as a file name, but the changes will not affect the value used for the file. This is because the value used for the file is set when the control file is processed. Since the rule file is processed after the control file, it is too late to alter the file name.
-V	Validates the control file for correct tagging and syntax, but does not process input or output files. This switch is useful for troubleshooting control files.
-W=*maxRecordSize*	For Postscript only, specifies the maximum record size, in KB, of a record present in any input files. This should only be used if records greater than 32k are present in an input file.
-Y	On mainframe systems, prior to version 6.6.2, Enrichment opened READ files as text files. This caused records read to have spaces truncated. Depending on how the file was allocated, carriage control bytes could have been interpreted rather than returned to the user. Currently, Enrichment opens READ files as binary in order to return the true contents of the file. Enabling this flag, will cause Enrichment to open READ files as text files and process them as it did prior to version 6.6.2.
-Z	For mainframe systems only, removes trailing spaces from records read by the READ function.

Return Codes

Enrichment return codes identify the success or failure of an engine run. The standard Enrichment return codes are as follows:

0 = Normal completion

2 = Successful completion with one or more information messages issued

4 = Successful completion with one or more warning messages issued

8 = Enrichment terminated with a severe error

Other return codes may be generated by the user code (the MESSAGE function) or the system run-time environments.

Using Field Values to Halt Processing

You can use the MESSAGE function to halt processing based on the value of a field. For example, if you want to halt processing if the account number is null, you could define a field called `%%ACCOUNT_NUM` and specify the following in your rule file:

```
if %%ACCOUNT_NUM = "" then
     message(1, S, "Account number is missing")
endif
```

This would generate a severe message (as indicated by the "S") which would halt processing. For more information on the MESSAGE function, see the *Enrichment Language Reference Guide*.

Performance Tuning

You can control most performance issues by using Environment tag group tags that are related to memory usage. Placing these tags in your control file allows you to manage the following aspects of memory usage:

Single document buffer controlled by the <SINGLEBUFFER> tag

Document sort buffer controlled by the <DOCBUFFER> tag

Temporary document storage buffer controlled by the <IO>, <FILEBUFFER>, and <MAXFILEBUFFERS> tags

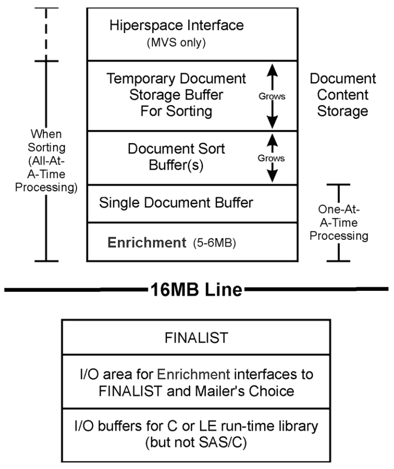
Mainframe Hiperspace controlled by the <IO>, <HIPERMAX>, and <HIPERTRANSFER> tags

These features are discussed in detail below. To determine the appropriate settings for these tags, look at the Enrichment report that is generated when you run your application. The Enrichment report will tell you how much memory was used. You can optimize performance by setting the initial memory block sizes to the reported levels using appropriate the Environment tag group tags.

> **Note:** Enrichment applications require enough memory to hold both the program and the contents of a single document. The Enrichment program and standard data elements typically require 5 to 6 MB of memory.

The following diagram illustrates memory usage.

> **Note:** On mainframe systems, Enrichment can run above or below the 16 MB line.


Hiperspace Interface
(MVS only)
Temporary Document Storage Buffer For Sorting
Grows
Document Content Storage
When Sorting (All-At-A-Time Processing)
Document Sort Buffer(s)
Grows
Single Document Buffer
One-At-A-Time Processing
Enrichment (5-6MB)
16MB Line
FINALIST
I/O area for Enrichment interfaces to FINALIST and Mailer's Choice
I/O buffers for C or LE run-time library (but not SAS/C)

Single-Document Buffer

The single-document buffer holds one entire document while Enrichment searches within it for fields. Enrichment allocates a block of memory for this buffer when the program starts. If Enrichment reads a document that does not fit into the buffer, it automatically increases the buffer size until the document fits. The single-document buffer will be at least as big as the largest document in the input.

Use the Environment tag group <SINGLEBUFFER> tag to control the initial size of the single-document buffer (in the range 1 KB to 2 GB) and its rate of growth, or growth factor. The growth factor is a percentage (in the range 1% to 200%) of the buffer size at the time it is being increased. Enrichment automatically increases the buffer size by the growth factor as necessary until the document fits. The default <SINGLEBUFFER> tag setting for initial buffer size is 32 KB. The default growth factor setting is 50.

For example, if you set <SINGLEBUFFER> initially to 10 KB with a growth factor of 50% and the current document is 20 KB in size, Enrichment increases the single-document buffer size first by 5 KB (50% of 10 KB) to 15 KB and then by 7.5 KB (50% of 15 KB) to 22.5 KB.

For complete information on using the <SINGLEBUFFER> tag, refer to the *Enrichment Language Reference Guide*.

Document Sort Buffer

The Document Sort Buffer stores an internal index and data about the documents. It is only used during all-at-a-time processing. Enrichment allocates a block of memory for the document properties buffer when the program starts. If the size of this buffer is insufficient, Enrichment automatically doubles it. Use the Environment group <DOCBUFFER> tag to control the initial size of the document properties buffer (in the range 1K to 2G-1K). Adjusting the initial size of the buffer minimizes the number of times that reallocation occurs.

For complete information on using the <DOCBUFFER> tag, refer to the *Enrichment Language Reference Guide*.

Temporary Document Storage Buffer

The Temporary Document Storage Buffer holds the contents of all documents. It is only used during all-at-a-time processing. By temporarily storing the documents during sorting, Enrichment can significantly reduce I/O and analysis time. On mainframe systems, the actual storage of these documents can be in memory, in VSAM files, in hiperspace memory, or in a combination of these. On Windows and UNIX systems documents can be stored in memory or on disk. To specify the storage location, use the Input tag group <IO> tag. For complete information on the <IO> tag, refer to the *Enrichment Language Reference Guide*.

The <FILEBUFFER> tag specifies the size of the memory blocks to acquire each time Enrichment requires more memory. If you set the size of the file buffers too low, Enrichment must frequently obtain more memory blocks during processing. The default <FILEBUFFER> setting is 200 KB. For complete information on using the <FILEBUFFER> tag, refer to the *Enrichment Language Reference Guide*.

The <MAXFILEBUFFERS> tag sets the maximum number of file buffers that Enrichment can acquire. Once the maximum is reached, Enrichment switches to either hiperspace or disk, based on the <IO> tag setting. You can set the <MAXFILEBUFFERS> tag to an integer in the range 1 to 10000. For complete information on using the <MAXFILEBUFFERS> tag, refer to the *Enrichment Language Reference Guide*.

Required Print Resources

Depending on your printing environment, Enrichment may require some of the following external print resources for processing:

- AFP environments: PAGEDEFs, FORMDEFs, overlays, page segments (PSEGs), and fonts
- Xerox environments: forms, fonts, IMGs, Job Source Language (JSL), and JDLs
- PCL environments: fonts and macros
- PostScript environments: fonts or other resources

Enrichment does not create or modify print resources (although Enrichment can change which resources a job uses). It typically uses the print resources for an existing application without change. However, if you wish to change the look of the document, you must identify or create print resources for the print applications.

AFP Print Resources

AFP printing systems, such as IBM's Print Services Facility (PSF), require that all copy groups and page formats for a single print job be contained in a single PAGEDEF and FORMDEF. You must ensure that these resources are available before printing.

Consolidated FORMDEFs and PAGEDEFs

When multiple document streams that use different PAGEDEFs or FORMDEFs are merged, a comprehensive PAGEDEF or FORMDEF must be created. This is accomplished by combining the information from these resources into a single resource.

With Page Printer Formatting Aid (PPFA), the process is fairly simple:

1. Combine the source code for various copy groups and place it in one FORMDEF.
2. Combine the source code for various page formats and place it in one PAGEDEF.
3. Process the tags with PPFA.

Note: If you are unfamiliar with PPFA or have not used the COPYGROUP or PAGEFORMAT commands before, refer to the *IBM Page Printer Formatting Aid/370: User's Guide and Reference*.

If you use Print Management Facility (PMF) or other menu-driven AFP resource generators, use the menus to generate new FORMDEFs and PAGEDEFs that contain the combined copy groups and page formats.

Xerox Print Resources

Metacode and DJDE use DJDE records to select the appropriate JDL, Job Descriptor Entry (JDE), or form for printing. Enrichment cannot modify JDLs or forms because they are maintained on the printer fixed disk, separate from the business application. However, Enrichment can add or change DJDE records to indicate the type of print resources to use.

PCL Print Resources

Some applications use macros to define forms or other objects that might appear on a page. You can load these macros into the printer permanently or include them at the beginning of a file. Enrichment cannot modify permanent resources but can change which resources are invoked.

PostScript Print Resources

Some applications use predefined objects. You can load these objects into the printer permanently or include them at the beginning of a file.

6 - Troubleshooting an Application

In this section

Troubleshooting an Application

If problems arise when you run the application, complete the following checks:

- Ensure that your script includes all files and the Enrichment module. Also ensure that the necessary files exist on the system.
- Ensure that print stream analysis coding in your control file is correct by following these steps.
  1. Create a small test input file with a known number of pages and documents. We recommend 10 to 100 documents—enough to get a representative sample.
  2. Check the Enrichment Report to ensure that Enrichment identified the correct number of documents and pages.
  3. Write one document to a specific output to make sure it emerges in the correct format with no missing parts. This can verify that you specified the <DOCUMENT> and <PAGE> tags correctly.
  4. Write a “trace” extract file or use the FOUND function to verify that you are collecting the proper data from the document. This confirms that fields are being selected properly. You can also write page numbers and other system variables to the output to ensure that the documents are being selected correctly.

     Note: Because it is common for print streams to change from time to time, you should build your application to anticipate changes to important parts of the print stream. We recommend that you use the FOUND function to detect when critical fields are not found during processing. You can also use this during development to see if any documents are unusual (for example, if fields are not in the predicted locations). Use the MESSAGE function to report errors.

- Verify that your Enrichment application does not contain any of the following common user errors:
  - The <DOCUMENT> tag specification is never met. Enrichment tries to read your entire input into memory as a single document. If you are running a small sample input, the Enrichment Report will show only one document found in the input. If you are running a large input, Enrichment will run out of memory and issue the error shown below.

    "PDR2001S Unable to acquire number bytes of memory to enlarge single document buffer."
  - The <DOCUMENT> tag location is above the <PAGE> tag target. As a result, Enrichment does not find documents correctly.
  - The <DOCUMENT> tag specifies a top-of-document location that is after the first complete page of the document.
  - Fields are improperly extracted from the document because of incorrect referencing.
- The record length of the output file is incorrectly specified:

  - Enrichment issues a warning indicating that "records were truncated" if the record length is incorrect. This results in incomplete documents, which typically cannot be printed.
  - If you are adding drawn barcodes to a document, you must set up the output file as variable length.

- The ZIP Code™ and the ZIP + 4® are not in the index file to control presorting as required. If you use CASS™ cleansing to obtain the ZIP Code™ and ZIP + 4®, the POSTNET™ barcode will be blank if the address was not correct. Therefore, you should not place the POSTNET™ barcode in the index file for all documents. If the POSTNET™ is bad, place the ZIP Code™ in the index (or put the original 5-digit ZIP Code™ and the ZIP + 4® from the POSTNET™, if available, in the index for all documents). Otherwise, invalid presorting may occur.
- The reject file has been confused with the residual file. During postal presort, it is often incorrectly assumed that the reject file is the same as the residual file. Residual documents are not normally deleted from the sorted index file during presort, but are instead placed at the end of the normal output stream. Rejected documents are those whose record were dropped by the program called in the Presort tag group so there is no corresponding record returned in the <OUTFILE1> file. Note that documents that fall into this category have no additional processing done on them, including <ADD> processing. If there is no <REJECTFILE> tag, then these documents will not be included in the output.
- The CASS™ cleansing parameters are not set correctly. This normally affects the percentage of mail that is verified.
- The end tag is left off of a tag group structure. This usually results in an error message stating that one or more tags are not valid for the tag group in question.
- The page size in the FORMDEF or PAGEDEF does not match the page size used by Enrichment.
- Variables are not initialized or reset properly in a rule file. This can cause erroneous results in which the variable retains a value from the previous document or the original value used for the variable is not what was expected. It is recommended that variables be initialized in the START: section of the rule file.
- An incorrect version of the C run-time libraries is used. Make sure you have the latest version of the C run-time libraries installed on your mainframe system.
- The errors PDR2000S and PDR2002S indicate that you have used all available memory. On mainframe systems, this may require an increase in the REGION size. These errors may indicate that Enrichment is reading a large file as a single document.
- Review the statistics at the end of a successful test run to get an idea of how you can tune the <SINGLEBUFFER> setting. An improper <SINGLEBUFFER> value can cause problems. Assuming the test file is representative of a production file, a suggested starting value for <SINGLEBUFFER> is 110% of the value used for the single document buffer as reported in the Enrichment report. Generally a growth factor of 50 is appropriate. The idea is to set the initial value large enough to hold the largest document and eliminate the need to make this area larger. The single document buffer is the only area that must be in contiguous memory.

Testing Performance

The amount of time Enrichment requires for processing depends on the functions performed, the system used, and other system activities concurrent with the run. You should test Enrichment to ensure that it completes processing in a time frame compatible with your print processing window. In some cases, Enrichment may actually save time by eliminating steps in your current process.

On mainframe systems, you should also consider whether your test jobs are getting the same priority as production jobs. In many cases, test jobs run slower because they are given lower system priority than production jobs. You should also check the level of your run-time library (IBM C/370 Runtime Library, SAA AD/Cycle Language Environment, or Open Edition AD/Cycle C/370 Language Support Feature). Fixes or newer versions of the run-time library may be available that would enhance your application's performance.

Notices